SQL AND NOSQL DATABASES FOR CYBER PHYSICAL PRODUCTION SYSTEMS IN INTERNET OF THINGS FOR MANUFACTURING (IOTFM)

A Thesis
Presented to
The Academic Faculty

By

David Gamero

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
School of Mechanical Engineering
Department of Mechanical Engineering

Georgia Institute of Technology

May 2021

SQL AND NOSQL DATABASES FOR CYBER PHYSICAL PRODUCTION SYSTEMS IN INTERNET OF THINGS FOR MANUFACTURING (IOTFM)

Thesis committee:

Dr. Katherine Fu
Mechanical Engineering
Georgia Institute of Technology

Dr. Christopher Saldana
Mechanical Engineering
Georgia Institute of Technology

Dr. Thomas Kurfess
Mechanical Engineering
Georgia Institute of Technology

Date approved: April 21, 2021

Luck is what happens when preparation meets opportunity.

Seneca

To My Beloved Family

ACKNOWLEDGMENTS

I would like to thank the members of my thesis committee for their help in preparation of this work – Katherine Fu, whose endless support and planning made this work possible, Thomas Kurfess, whose extensive industry experience kept me on track and grounded, and Christopher Saldana, who was invaluable in helping articulate my ideas into a line of inquiry.

Special thanks are due to Andrew Dugenske, whose expertise consistently revealed new blind spots to explore, and who was always available to talk through even my most outlandish ideas. Joshua Von Holtz and Walker Poole were invaluable both as friends and sounding boards. I could never thank Jenny Guzdial enough for her unending encouragement pushing me to be the best version of myself.

This work was funded by the U.S. Department of Energy, under award DE-EE-0008303. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

ACID Atomic, Consistent, Isolated, and Durable

AMQP Advanced Message Queuing Protocol

API Application Programmer Interface

AWS Amazon Web Services

CoAP Contrained Application Protocol

CPPS Cyber Physical Production Systems

CPS Cyber Physical Systems

CPU Central Processing Unit

DBMS Database Management System

EC2 Elastic Compute Cloud

GCP Google Cloud Platform

HTTP Hypertext Transfer Protocol

IaaS Infrastructure as a Service

IAM Identity and Access Management

IIoT Industrial Internet of Things

IoT Internet of Things

IoTaaS IoT as a Service

IoTfM Internet of Things for Manufacturing

IP Internet Protocol

JSON Javascript Object Notation

MQTT Message Queuing Telemetry Transport

NIST National Institute of Standards and Technology

OLAP Online Analytical Processing

OLTP Online Transaction Processing

OSI Open Systems Interconnection

PaaS Platform as a Service

Pub/Sub Publish-Subcribe

QoS Quality of Service

RAM Random Access Memory

RDS Relational Database Service

SaaS Software as a Service

SCP Secure Copy Protocol

SQL Structured Query Language

SSH Secure Shell

SSL Secure Sockets Layer

TCP Transmission Control Protocol

TLS Transport Layer Security

TSDB Timestream Databases

UDP User Datagram Protocol

XML Extensible Markup Language

SUMMARY

The proliferation of low-cost sensors and industrial data solutions have continued to push the frontier of manufacturing technology. Machine Learning and other advanced statistical techniques stand to provide tremendous advantages in production capabilities, optimization, monitoring, and efficiency. The tremendous volume of data gathered continues to grow, and the methods for storing the data are critical underpinnings for advancing manufacturing technology. This work aims to investigate the ramifications and design trade offs within a decoupled architecture of two prominent Database Management Systems: SQL and NoSQL. A representative comparison is carried out with Amazon Web Services (AWS) DynamoDB and AWS Aurora MySQL. The technologies and accompanying design constraints are investigated, and a side-by-side comparison is carried out through high-fidelity industrial data simulated load tests using metrics from a major US manufacturer. The results support the use of simulated client load testing for comparing latency and throughput of database management systems as a system scales. As a result of complex query support, MySQL is favored for higher order insights, while NoSQL can reduce system latency for known access patterns at the expense of integrated query flexibility.

CHAPTER 1
INTRODUCTION

1.1 Introduction

The Internet of Things (IoT) brings enhanced productivity to industrial manufacturing environments. The next big leap in manufacturing technology is represented by the German strategic initiative Industry 4.0, in which IoT, Big Data and Service fundamentally alter production as described by Kagermann [1]. Also referred to as the Industrial Internet of Things (IIoT), or Internet of Things for Manufacturing (IoTfM) [2], this revolution holds the potential to create a tremendous surge in manufacturing productivity, driven by real-time access to vast droves of previously inaccessible data in a granular, non-cost prohibitive format.

1.2 Motivation

The growing intersection of automation and manufacturing has given rise to a proliferation of new sources of information, allowing increasingly sophisticated analysis of industrial data. As the space grows, the methods, machines, and formats of analyzing the data range from low-cost distributed sensors to specialized machine learning compute clusters in the cloud. The backbone of this revolution is access to data that was previously cost-prohibitive to acquire. Declining data storage costs allow historic records to be easily archived for future analysis, and low latency IoT services can provide crucial glimpses into live machine states across the world.

In this emerging space, the need to understand data storage technologies, and their tradeoffs, is critical to every process that operates on that data. Generalized performance metrics allow comparisons between Database Management Systems. As data read and

write access is a fundamental activity in digital manufacturing, architectural decisions made, such as selection of a Database Management System (DBMS), have cascading effects on the performance, scalability, and design constraints for entire installations of industrial sensor systems. Understanding the implications of different query languages and data storage technologies, along with their relative compatibility with respect to industrial sensor installations in manufacturing settings, will be necessary in order to meet the full potential of the digital industrial revolution. This work aims to characterize architectural and design differences between Structured Query Language (SQL) and NoSQL databases. An implementation comparison is carried out using AWS cloud services and simulated client loads with a model seeded from over a year of live IoT data from instrumented assets at a major US manufacturing firm.

CHAPTER 2
BACKGROUND

Several technologies, protocols, and models are used in the field of Cyber Physical Systems (CPS). This chapter introduces these layers in successive order as each technology builds upon previous ones. Additionally, proposed architectures for Cyber Physical Production Systems (CPPS) are introduced. Several database technologies and types are described, and existing work investigating databases in both generalized and IoTfM use cases is summarized. Next, one area of investigation is highlighted, and the research questions for this work are introduced.

2.1 Cyber Physical Systems

At the intersection of cyber systems and physical systems, CPS have become increasingly ubiquitous, driven by plummeting hardware costs and commoditized network access in manufacturing environments. As asserted by Mourtzis et al., the amount of data generated as low-cost sensors proliferate is rapidly growing [3]. The work presented in this thesis is highly relevant to the design and data management of CPS.

2.2 Internet Standards and Protocols

In order for IoT to exist, several networking and connectivity technologies are necessary. IoT is built on existing Internet communication technologies and protocols that facilitate data transmission and access. This section introduces terminologies and standards used in this work, including conceptual models as well as specific protocols and paradigms.

2.2.1 OSI 7 Layer Model

The Open Systems Interconnection (OSI) 7-layer model describes the data, protocols, and applications in network communication [4, 5]. Layer 1, the Physical Layer, includes physical connections, such as wires via electrical or fiber optics and wireless connections. Layer 1 also includes hubs for connecting elements within this layer. Layer 2, the Data Link Layer, includes higher level connections between devices using standards like Ethernet, switches and bridges. The third Layer, the Network Layer, facilitates routing capabilities via protocols like Internet Protocol (IP) [6]. Layer 4 includes protocols for end-to-end connections in the Transport Layer, and Layer 5 adds Application Programmer Interface (API) capability and sockets in the Session Layer. Layer 6 is the Presentation layer, adding Secure Sockets Layer (SSL) encryption, syntax, and formats like Javascript Object Notation (JSON) [7] and Extensible Markup Language (XML) [8] for data interchange. The highest layer is the Application Layer, which is Layer 7, and it contains protocols like Hypertext Transfer Protocol (HTTP) [9] and Message Queuing Telemetry Transport (MQTT).

2.2.2 TCP UDP

User Datagram Protocol (UDP) is a protocol that builds upon IP to offer a messaging protocol with transaction support. UDP transactions don't include acknowledgement or ordered data stream delivery [11]. Transmission Control Protocol (TCP) is a "highly reliable host-to-host protocol" with segment acknowledgement, ordered data streaming, and host-to-host connections as shown in Figure 2.2 [12].

2.2.3 MQTT

MQTT "is designed as an extremely lightweight publish/subscribe messaging transport that is ideal for connecting remote devices with a small code footprint and minimal network bandwidth" [14]. As shown in Figure 2.3, MQTT makes use of a broker, acting as a message bus, to transfer messages from publishing clients to receiving clients by topic.

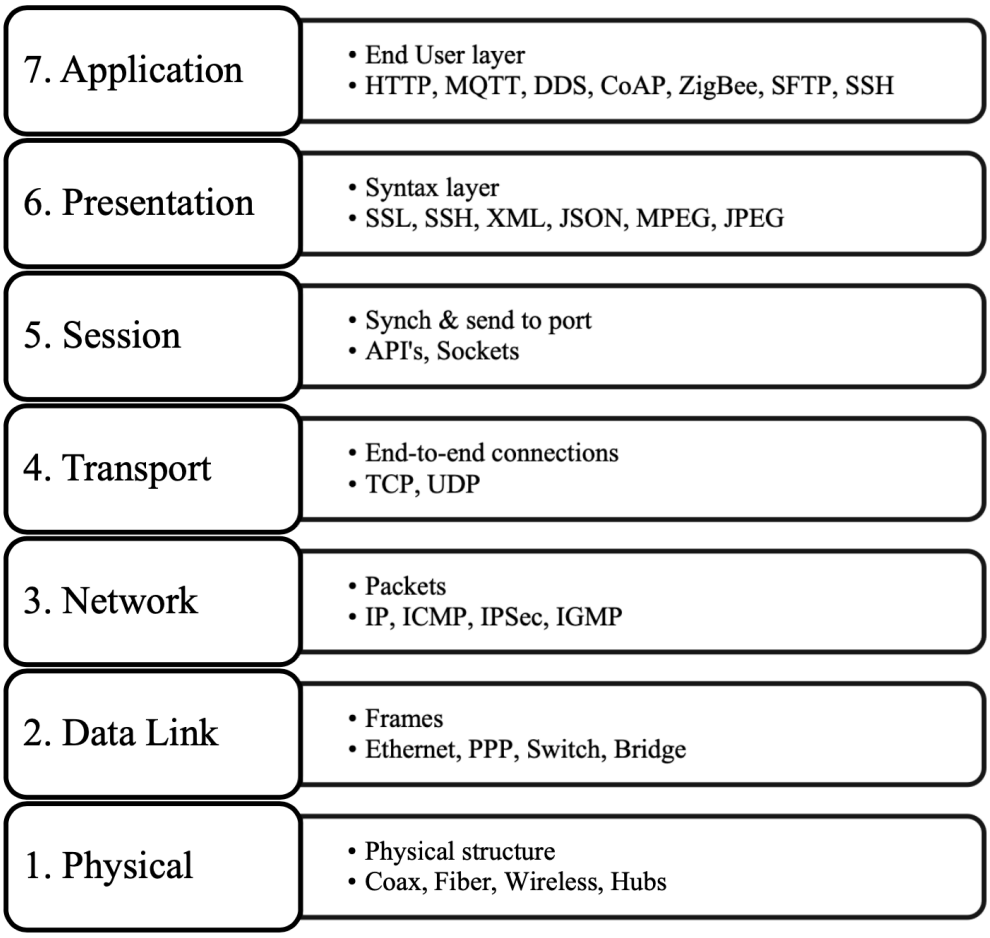

Figure 2.1: The ISO 7-layer model describes successive layers of network communication. Figure from [10]

Topics are arbitrarily designated within an installation to segment information and allow finer subscription granularity. MQTT has found wide use in IoT, as it allows for flexible architectures with multiple clients and subscribers without heavy computational requirements or overhead [15] [16].

Alongside HTTP and MQTT, protocols such as Contrained Application Protocol (CoAP) [17] and Advanced Message Queuing Protocol (AMQP) [18] are used in the IoT space. For IoT systems, each of these protocols has advantages in compatibility, message size, architecture, and encryption. MQTT is the default protocol for the major cloud providers, including AWS, Google Cloud Platform (GCP), and Microsoft Azure [19].

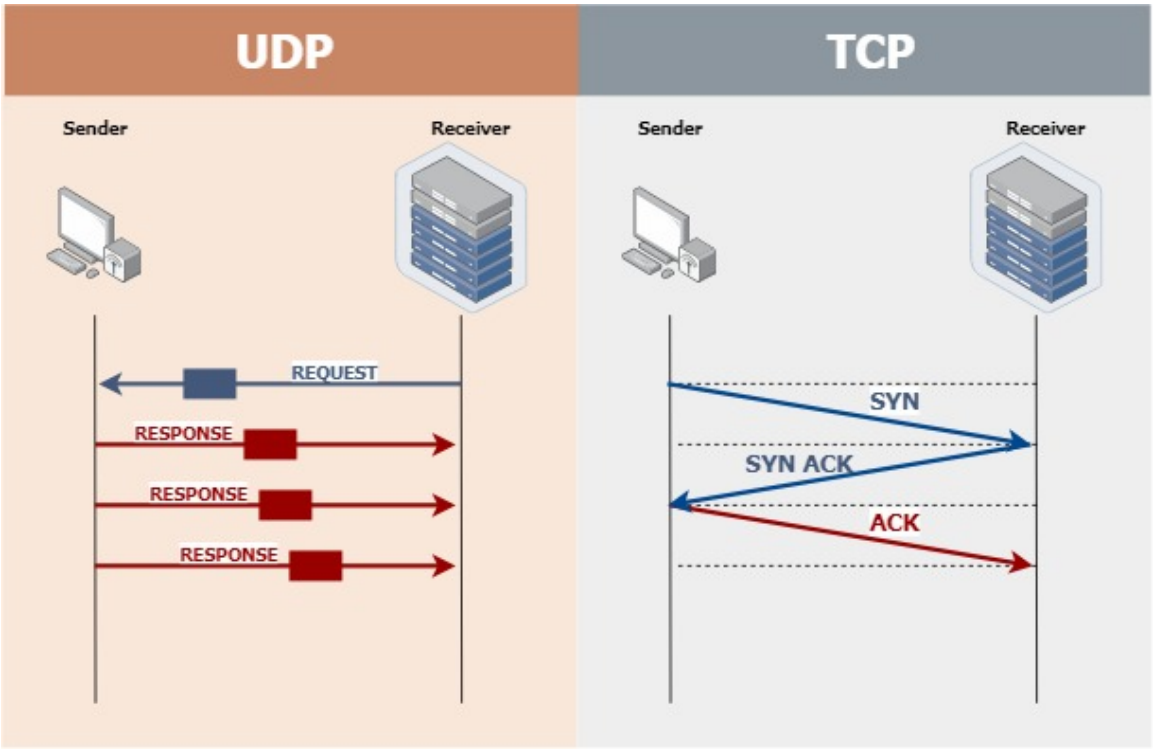

Figure 2.2: TCP and UDP Protocols [13]

Quality of Service

MQTT supports transmission of messages with three Quality of Service (QoS) levels. QoS level 0 does not include message receipt acknowledgement, and messages are only transmitted once. This QoS level has the least overhead, and the tradeoff is that messages are not guaranteed to be delivered. QoS level 1 guarantees a message is received at least once, but could be delivered multiple times. A copy is kept by the sender until the message receipt is confirmed. QoS level 2 guarantees a message is received exactly once, and includes coordinated transmission in which both ends acknowledge transmission and receipt of the message [14].

2.3 Cloud Computing

The growth of Cloud Computing is exemplified by a ”shift in the geography of computing”, in which software is executed on remote computers in data centers accessed through the

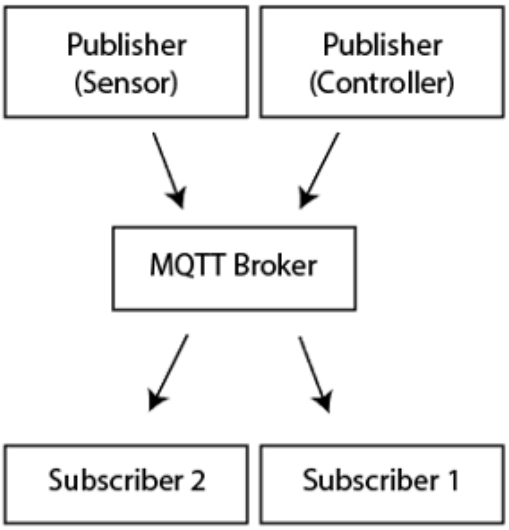

Figure 2.3: A message broker acts as a bus between subscribers and publishers in the MQTT protocol

internet [20]. The tasks involved in managing hardware are handled by Cloud Providers who offer incremental pricing for access to a set of centralized servers. As defined by The National Institute of Standards and Technology (NIST), the Cloud Computing model allows various clients to share computing resources as services. A main tenet is the ease with which clients can easily change service requirements in a low-cost way [21]. The basic services provided include software, platform, and infrastructure, provided in a user-centric and task-centric way [22]. Major providers include AWS [23], Microsoft Azure [24], and GCP [25], with each offering a variety of services [26].

2.4 Architecture

Data can be transmitted to the cloud from sensors installed on assets, transmitted locally or used on-premise. In the decoupled architecture proposed by Nguyen and Dugenske [2], gateways have the potential to aggregate data into a single connection from multiple discrete sensors. The sensor data is acquired using low-cost hardware, and then the data are transmitted over a Publish-Subcribe (Pub/Sub) architecture to a message bus where data

consumers can receive live data. One of the consumers proposed is an archiving service that inserts data into a database to keep historical records. More distributed proposed architectures include a network connection for each individual sensor.

Bonci et al. [27] proposed an architecture that includes running lightweight SQL databases on each sensor to further decentralize the data. This approach creates duplicate records across the databases, and heavily relies on execution of stored procedures to transmit the new data to synchronize all the independent databases. This work recognized the potential of MQTT as a data distribution and Pub/Sub for distribution IoT payloads.

2.5 IoT as a Service

The prevalence of IoT has led to development of standardized offerings from major cloud providers that leverage MQTT to create managed IoT deployments and integrations as a service, known as IoT as a Service (IoTaaS). AWS IoT core is a Software as a Service (SaaS), which can be integrated with Infrastructure as a Service (IaaS) and Platform as a Service (PaaS) offerings from Amazon Web Services. Similar to offerings from Azure IoT and Google Cloud IoT, each offer Pub/Sub using MQTT. Client connections in AWS IoT Core are capped in data message transfer frequency by AWS; for example, a single AWS IoT core client is limited to 100Hz message frequency. Messages that exceed the client message publish quota are discarded. This limits the number of shared connections on a single gateway; however, additional connections are trivial to implement, bypassing this limitation.

2.6 Edge Computing

Edge computing moves the processing from a centralized cloud to distributed deployed systems closer to monitored assets. Moving computing to the edge trades larger data scope for reduced latency, potential decrease in failure modes and more distributed compute loads. Edge computed results can be calculated on-site or on-device, reducing dependence on a

connection to a centralized or cloud network. Also, by computing higher order metrics on-premises or on-device using edge computing, the data transmitted to the message bus can be reduced if only the higher-order metrics are to be retained for archival purposes in the cloud.

2.7 5Cs Architecture Model of IoT

Within the 5Cs Architecture Model proposed and discussed by Lee et al. [28], and Monostori et al. [29], every level is built upon increasingly complex and auto-correlated analysis of underlying sensor data, as depicted in Figure 2.4. Beginning at the base level in which a sensor network establishes communication, the Data Conversion, Cyber, Cognition and Configuration levels delineate progressively removed and abstract insights.

The architecture is built in an unopinionated way towards data storage but is influenced by latency and throughput limitations of all underlying technologies. As we rise through the layers, the complexity of analysis and the processing power necessary to derive insights increases. Each layer performs additional transformations and calculations on the computed outputs of the layers below it, culminating in Layer 5, in which machines can self-configure, self-adjust, and self-optimize.

2.8 Big Data

Data from Industrial IoT can grow to volume, velocity and variety consistent with Big Data [30]. For analyzing Industrial Big Data, the same techniques that non-industrial Big Data utilizes are applicable. Use of Data Lakes to store a combination of structured and unstructured data for archival analysis and insights leads to use of cluster-based scalable solutions, such as MapReduce, Hadoop, and Apache Spark. While ideal for large workloads, they require specialized implementations.

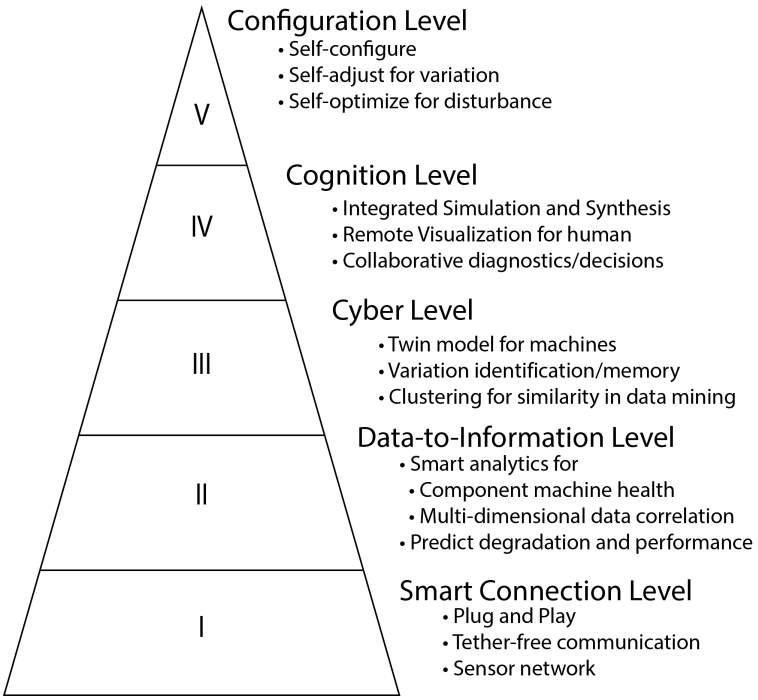

Figure 2.4: The 5Cs Pyramid depicts levels of Cyber Physical architecture. This figure is adapted from [28]

2.9 ACID Transactions

A set of properties, referred to as Atomic, Consistent, Isolated, and Durable (ACID), are desirable in database transactions as they ensure data integrity and validity in individual logical transactions. Atomicity is the first property, and it ensures that a single transaction entirely fails or succeeds without intermediate states, even with interruptions or errors. This prevents partial updates. Consistency maintains data validity by constraining transactions exclusively to states that comply with all existing data rules. It guards against transactions that violate constraints, such as foreign keys. Isolation ensures that transactions that occur concurrently do not affect the database in any ways different than if they were executed one after the other. Durability is the property that requires data is protected from volatility once a transaction is committed. Data will not mutate once in a committed state unless modified by a transaction [31].

2.10 OLAP and OLTP

There are two primary data system types, Online Analytical Processing (OLAP) and Online Transaction Processing (OLTP). OLTP is built towards high transactional speed direct transactions, like data insertion, updating, and deletion. Within most SQL systems, OLTP is used to interact with data in an ACID way for direct data manipulation and retrieval, and it includes reading or writing data in normalized form. OLTP commonly stores data in rows that are indexed, quickly accessible on an individual basis, and stored in blocks on disk. OLAP systems use analytical queries to extract more complex, metrics and relationships from data sources. Data warehouses are a common OLAP use case, and more frequently include de-normalized data. OLAP systems are designed for statistics and aggregation, as they use column compression and columnar stores to access information across multiple rows faster. Current NoSQL offerings leverage OLTP for models such as document stores and key-value stores [32, 33].

2.11 Database Management Systems

The prevailing architectural models for CPPS store the data in databases or data stores and use queries to extract insights from each layer of data to produce higher order information [34]. Some proposed architectures [27] suggest SQL databases as an option, while others are unopinionated towards database technology. Major types of database systems are shown in Figure 2.5

2.12 Configuration

SQL databases differ from NoSQL databases in a series of critical ways. SQL databases have a rigidly defined schema, which requires that the data fields be known in advance to configure the database to store records prior to receiving them [36]. SQL databases can be configured on a single server with optional additional read replicas, or in a sharded cluster

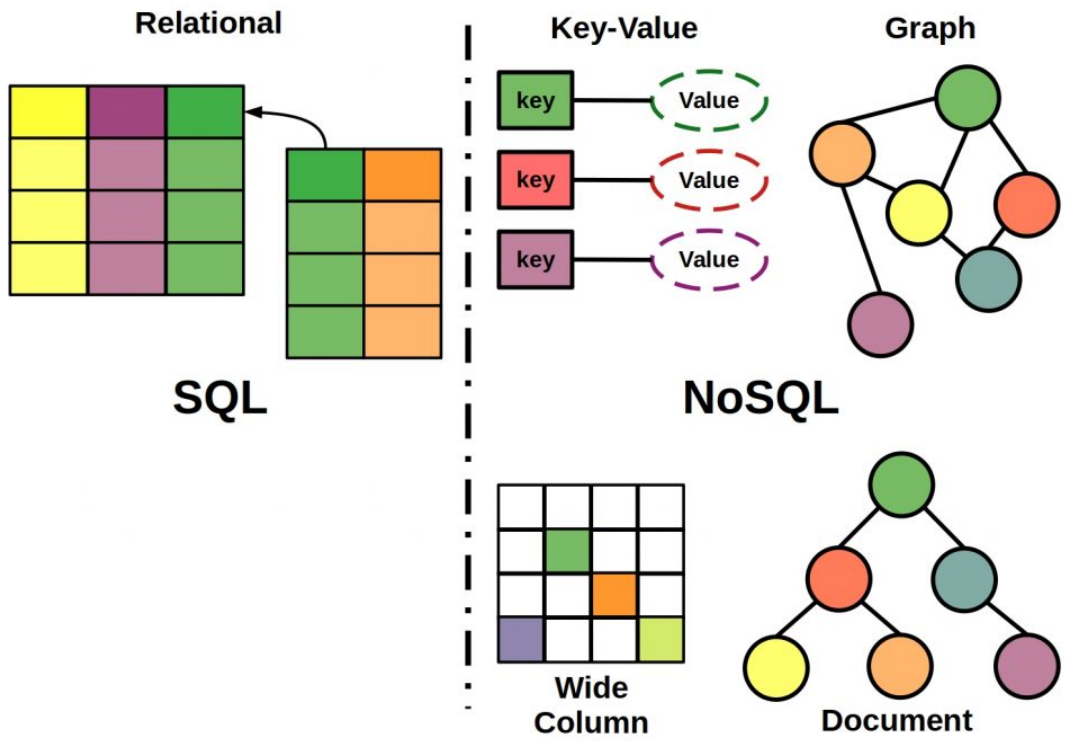

Figure 2.5: Major database designs for SQL and NoSQL Databases. This figure is adapted from [35]

configuration. SQL databases support ACID transactions, creating a reliable enterprise data storage system. The storage-optimized nature of SQL enables flexible querying of rigidly defined data schemas that may require complex relationships [37].

2.13 AWS Relational Database Service

AWS Relational Database Service (RDS) offers single-master and multi-master configurations for MySQL compatible Aurora database clusters. In single-master configuration, write throughput scales vertically with hardware, in that upgrading a dedicated MySQL instance can improve performance, but additional servers cannot be added to achieve a similar increase. Single-master clusters have a single write instance and multiple read replicas that increase data availability. Multi-master configurations allow continuous availability via writing across multiple instances in the cluster. For the purposes of this thesis, an Aurora

MySQL single-master cluster with a single instance was used.

2.14 NoSQL

NoSQL databases are frequently key-value or document stores in which a primary key and/or sort key are used to retrieve specific records without a rigidly defined table schema. Records can be inserted with arbitrary fields, and data relationships, such as foreign keys, are not strictly enforced. Data fields and relationships do not need to be known in advance when writing records to a NoSQL database. Additionally, NoSQL data are typically less normalized, leading to duplicate records that must be maintained. These properties are a result of the core architectural difference by which NoSQL achieves horizontal scalability: key partitioning. NoSQL databases can scale horizontally using commodity hardware while maintaining fast query speeds by eliminating the need to join data at query-time. For a NoSQL database to support the high read-write speeds, it must be designed with advanced knowledge of the query structures and access patterns that will be used for retrieving records. This structure is well-suited for high-traffic use with known access patterns, but becomes unwieldy with poorly defined access patterns.

For all the previously proposed architectures, data are the conduit by which increased instrumentation is converted into valuable insights and increases in manufacturing productivity.

2.15 SQL and NoSQL Performance

Comparisons of SQL and NoSQL databases have confirmed expected results for generalized and IoT use cases: the single-transaction latency and write speed of NoSQL databases has been found to increase at a lower rate with increasing transaction volume and frequency when compared to SQL databases in IoT applications [38, 39]. The ability to distribute disk writes across an arbitrary number of commodity instances makes NoSQL write performance excel compared to single-master SQL. The distributed data across instances results

in rigidly pre-defined access patterns; therefore, the flexibility of queries is dramatically restricted in NoSQL compared to SQL. Both databases have scalable implementations that sacrifice transaction consistency to achieve higher throughput [40].

2.16 IoT as a Service

Current IoT as a Service offerings can have zero-code integrations for writing received records on MQTT topics to both SQL and NoSQL databases, with integrated schema-generators for SQL and automatic primary/sort key configuration for NoSQL. Setup and client/device onboarding can be automated as well, with granular permissions and integrations available on major cloud providers.

Amazon Web Services offers IoT Core, an MQTT broker platform for IoT. Messages relayed to IoT Core are secured using a Secure Sockets Layer (SSL)/ Transport Layer Security (TLS) X.509 certificate [41], and are specific to an AWS account ”thing” under which the certificate is issued. Architecture norms use a single certificate per connected device to enable granular permissions and certificate authorization via attached AWS Identity and Access Management (IAM) roles that specify permissions at a device level. Permissions include topic Pub/Sub access and authorization.

2.17 Timestream Database Systems

Specialized Timestream Databases (TSDB) exist but are still in the developmental stage. They lack widespread adoption and integration with legacy systems and can be inflexible [42]. There is a fragmented offering of TSBD services, but they are still experimental in nature; they can be engineered to deliver higher compression for timeseries data, but are queried in similar methods to either NoSQL or SQL databases. The performance of timestream databases is not investigated in this work.

2.18 Data Streaming

Other tangential data pipeline management systems are Data Streaming services, such as Apache Kafka [43, 44] and AWS Kinesis. These platforms are designed for large-scale event streams and facilitate live analytics. Data streaming services function as an additional data routing layer that regularly includes a destination, such as a Database Management System; data streaming services act as a buffer and data conduit that aggregates data at a high rate while allowing transformations to be performed live on the data. Throughput and latency in these systems exhibits promising results in IoT applications, such as traffic monitoring [45]. Since they are not databases, they are not examined further in this work.

2.19 Summary

In summary, work done in the CPS field is inextricably linked with data storage and transmission technology. As the field grows, the volume of industrial data swells, which will create new challenges of data processing scale. Generalized work in distributed computing and database throughput has been extensively applied to manufacturing, but work leveraging the narrower subset of constraints for data architectures in manufacturing settings is limited. Proposed IoT date architectures are diverse and numerous, yet manufacturing data processes can be deeply entrenched in momentum as they grow beyond the proof of concept stage.

2.20 Research Questions

This work aims to address these needs with the following research objectives:

1. characterizing the scalability behaviors of NoSQL and SQL databases in the context of existing CPS and CPPS frameworks as measured by latency

2. enumerating scaling factors and bottlenecks encountered during synthetic load tests

seeded with data from a major US manufacturing firm

In order to investigate these questions, a test methodology is proposed, and evaluated with respect to existing CPS and CPPS frameworks. By extracting metadata parameters and statistics from an authentic manufacturing data set, the accuracy of existing generalized work can be examined with respect to the more rigidly defined use case of manufacturing IoT data.

2.21 Hypotheses

The generalized database results from established work will hold true in the narrow use case of IoTfM installations in that NoSQL will experience significantly greater throughput and lesser latency as compared to MySQL.

CHAPTER 3
METHODOLOGY

The methods proposed in this section aim to detail a high fidelity stress-testing architecture and its application in characterizing the scaling behaviors of an IoTfM installation using either a NoSQL or SQL database in AWS. The integration of statistics extracted from an active IoTfM installation is used to seed higher-fidelity simulated clients. First, the data set itself is introduced, followed by the steps by which representative statistics were extracted. These statistics capture distribution and average information for two key metrics in this analysis: latency and throughput. The latency corresponds to the full duration from message transmission over MQTT to the time at which the record is written into a final database. Throughput corresponds to the rate at which messages are transmitted through the system. Depending on the architecture of the IoTfM system, this can include many intermediate steps. In this work, the decoupled architecture proposed by Nguyen and Dugenske [2] will be examined. Next, the process for creating simulated clients for load testing purposes is detailed. The software and test plan configuration are introduced The simulated clients are used to stress test two different databases, AWS Aurora MySQL and DynamoDB.

3.1 Data

Data were collected from a large US manufacturing firm to determine the characteristics of the Cyber Physical Production systems that had been instrumented and running on an active production floor for over a year. Data were queried from a MySQL table of over 100 million records spanning 57 assets. The MySQL message archive table contains a full history of MQTT payloads and transmission timestamps for 18 months of the assets' activity from February 2019 to October 2020.

The system that collected the data followed the decoupled architecture proposed by Nguyen and Dugenske [2] in which data was received over MQTT and written to an AWS RDS MySQL instance. Non-production assets were detected via a flag on the associated data and excluded from this analysis, as they were used for testing the system and are not representative of an active Industrial IoT data system. SQL Queries were used to exclude test assets by using a WHERE clause to exclude data in a subquery.

The mean and standard deviation of the data payload size for each instrumented asset and the overall dataset were calculated to serve as representative samples. The mean and standard deviation of the frequency of message transmission for each asset was calculated. The mean and standard deviation for the message transmission frequency was also calculated for the entire data set as calculated with an arithmetic mean shown detailed in Equation 3.1 and sample standard deviation detailed in Equation 3.2. This value was used in conjunction with the data payload statistics to create a test plan in Apache JMeter.

$$\frac{1}{n}\sum_{i=i}^{n} x_i \tag{3.1}$$

$$S = \sqrt{\frac{\sum_{i=1}^{n}(X_i - \bar{X})^2}{(n-1)}} \tag{3.2}$$

Sample standard deviation S was calculated using values in the sample set X, the sample set mean $\bar{X}$ and the number of samples n.

3.1.1 Data Set Schema

Full copies of received MQTT messages were archived in a table along with metadata about the message. Columns used in this work and their datatypes are listed in Table 3.1. For the purposes of this work, the latency and throughput are the most relevant metrics. Average throughput can be calculated directly from the dateTimeRecieved column through

straightforward manipulation of an arithmetic mean, as shown in Equation 3.3.

$$\frac{1}{t_{end} - t_{start}}|X| \tag{3.3}$$

Where t_{start} and t_{end} are sample time bounds, and X is the set of recieved MQTT messages recieved between those bounds inclusively, Equation 3.3 is an expression for average throughput.

Table 3.1: Manufacturing Data Set Messages Schema

Column	Type	Note
Id	int(11)	autoincremented unique identifier
dateTimeReceived	timestamp	UNIX timestamp
topic	varchar	Slash-delimited MQTT Topic
payload	varchar	Stringified Payload JSON object
assetId	varchar	Asset Unique Identifier

3.2 Simulated Clients

In order to run load tests, client assets were simulated using Apache JMeter and integrating statistics taken from the data set to reproduce high-fidelity load scenarios. Apache JMeter is an open-source load-testing and performance-measuring application built using Java [46]. The software runs on the Java Virtual Machine, and supports plugins for interfacing through various protocols including MQTT.

Apache JMeter version 5.3 was used to simulate clients in an AWS Cloud Environment, transmitting messages from an Elastic Cloud Compute (EC2) instance to the AWS IoT Core service endpoint, as depicted in Figure 3.1. One client refers to a single sensor, instrument, or data stream source attached to a manufacturing asset. All experiments were run in the us-east-1 region. JMeter was executed from an AWS EC2 instance running the Amazon Machine Image $amzn2-ami-hvm-2.0.20200904.0-x86_64-gp2 sizedasat3.medium$.

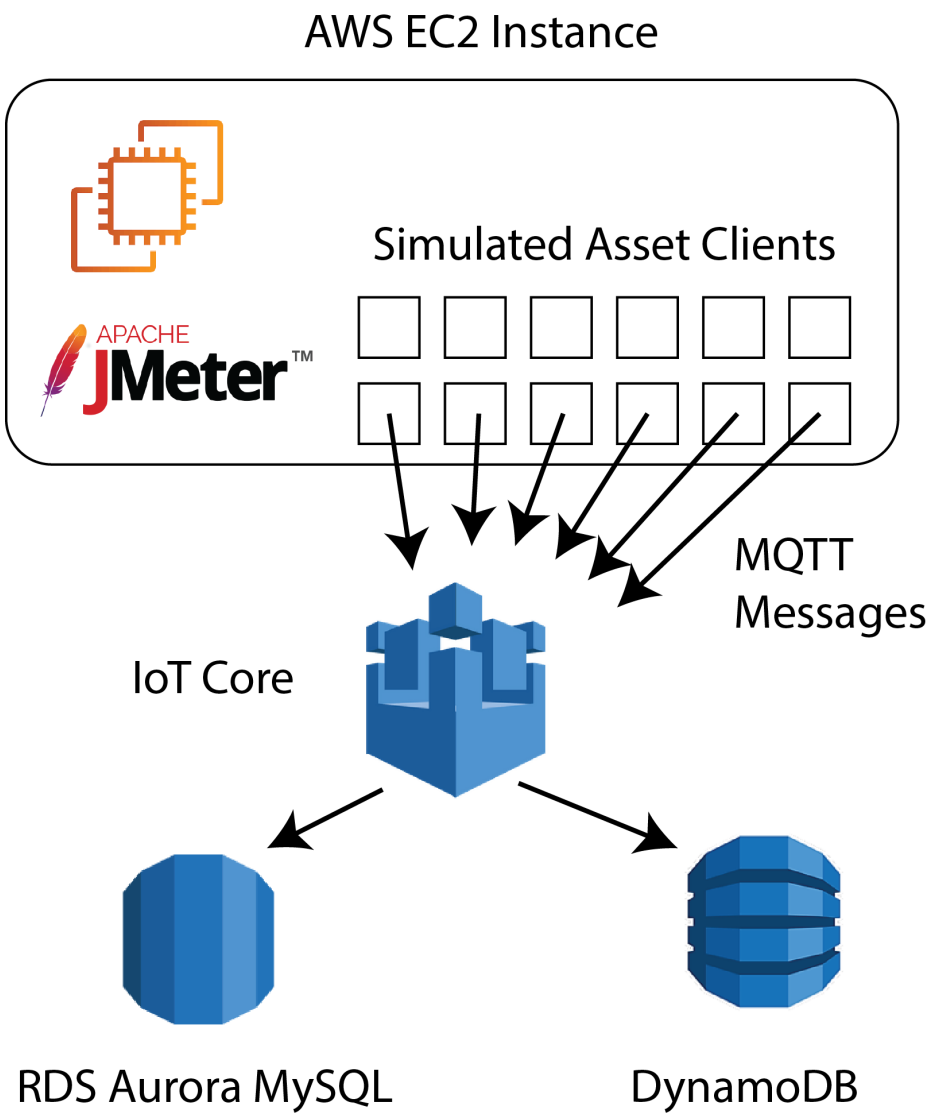

Figure 3.1: The virtual test bench leverages simulated asset clients in Apache JMeter running on an EC2 instance to create high-fidelity loading conditions for AWS IoT Core. IoT Core relays messages to DynamoDB and RDS Aurora MySQL database management systems.

3.2.1 MQTT plugin

The mqtt-jmeter plugin from xmeter-net was used to enable MQTT capabilities within the JMeter stress testing tool. A test plan file was created using the Graphical User Interface (GUI) mode, and then executed in command-line mode using flags and writing outputs to log files to ensure optimal testing performance.

3.2.2 Test Plan

The JMeter test plan was structured with a thread group containing three main stages: the MQTT Connect, Message Loop Logic Controller, and MQTT Disconnect. An Aggregate Report Listener and Summary Report Listener were used to collect results after the thread group during test plan development. Within the Message Loop Logic Controller, a Constant Throughput Timer was used to generate traffic, and the timer was configured using parameters extracted from the data set. A Gaussian Timer was added to introduce noise and configured using the parameters extracted from the data set as well. The test plan hierarchy during development is shown in Figure 3.2.

Within the JMeter Test plan, several fields use command line parameter substitution to allow the test plan variables to be modified by passing parameters as flags. This is used later in bash scripting for sweeping the variable space and automating the execution of tests with varied initial conditions. For example: "${P(clients,1)}" substitutes the value of the "clients" parameter from the command line flag, which is passed to Jmeter after the flag of the same name prefixed by the letter "J". In the following command, the number of clients is set to 10, which is parsed when the command is run, and substituted into the test plan for each instance of the parameter retrieval.

```
$ ./jmeter -n testplan.jmx -Jclients=10
```

The test plan was executed via the command line, with two primary parameters passed as flags, $clients$ and $duration$.

Table 3.2: Thread Group Configuration

Setting	Value	Type
Action to be taken after a Sampler error	Continue	radio
Name	<arbitrary >	text
Comments	<arbitrary >	text

Table 3.3: Thread Group Properties

Setting	Value	Input Type
Number of Threads (users)	${__P(clients,1)}	radio
Ramp-up period (seconds)	0	number
Loop Count	1	number
Infinite (Loop Count)	false	radio
Same user on each iteration	true	radio
Delay Thread creation until needed	false	radio
Specify Thread lifetime	false	radio

3.2.3 Thread Group

Within Apache JMeter, a test plan was created with a single thread group. The thread group was configured with attributes detailed in Table 3.2 and Table 3.3. The thread group was the singular highest level component in the test plan hierarchy, and it contained the MQTT Connect Sampler, Runtime Controller, and MQTT DisConnect Sampler.

3.2.4 MQTT Connect

The MQTT Connect Sampler was added to the Test Plan as the first child element to the thread group. This element was executed first within each thread in the test plan. It was configured for compatibility with AWS IoT Core. The certificate .p12 file that is supplied was downloaded from IoT Core after issuing a new certificate through the ”Add a new Thing” onboarding panel in AWS IoT. The certificate was granted read and write permissions for an arbitrarily named MQTT topic that was pre-specified for the duration of all load testing trials. The MQTT Connect Sampler was configured as detailed in Table 3.4

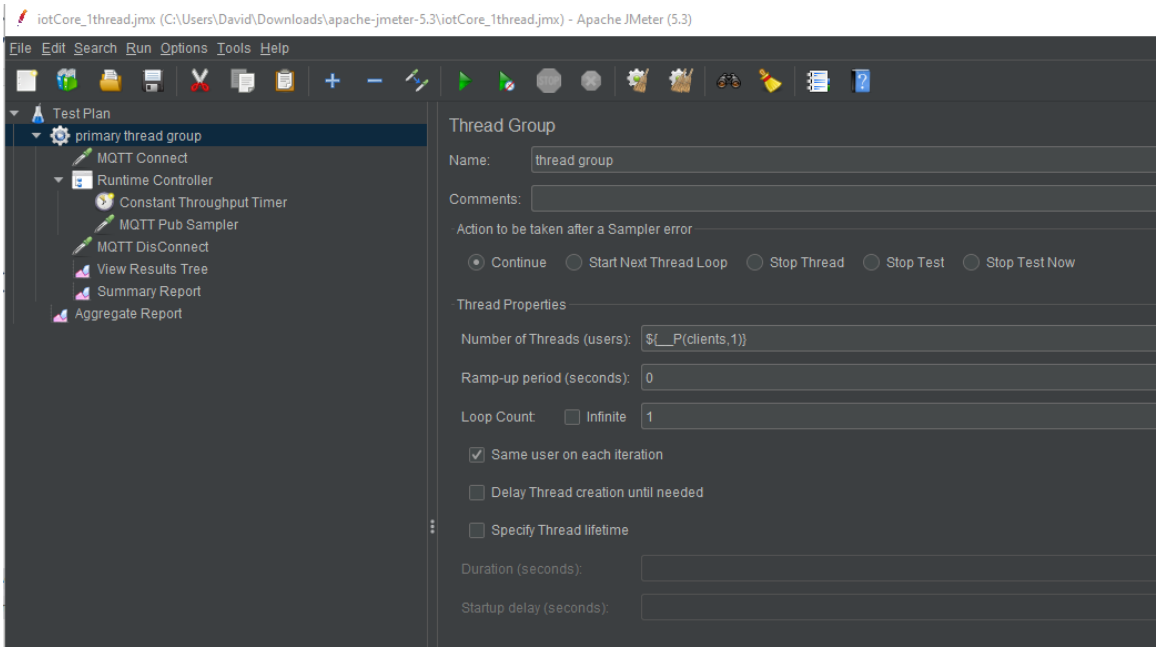

Figure 3.2: Test Plan Thread Group

Table 3.4: MQTT Connect Sampler Properties

Setting	Value	Input Type
MQTT connection	AWS IoT Core end-point address	radio
Port number	8883	number
MQTT Version	3.1	dropdown
Timeout(s)	10	number
Protocols	SSL	dropdown
Dual SSL Authentication	true	radio
Client Certification (*.p12)	Certificate from AWS IoT Core	file
Secret	<intentionally blank >	text
User name	<intentionally blank >	text
Password	<intentionally blank >	text
ClientId	conn	text
Add random suffix for CliendId	true	radio
Keep alive(s)	300	number
Connect attampt (sic) max	0	number
Reconnect attampt (sic) max	0	number
Clean session	true	text

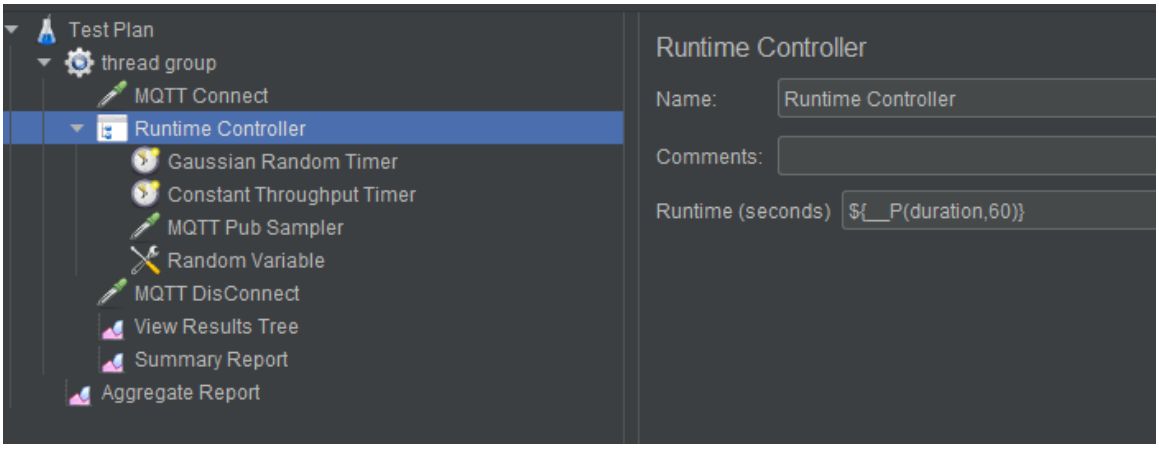

Figure 3.3: Runtime Controller Configuration

3.2.5 Runtime Controller

The runtime controller contained the main execution loop as shown in Figure 3.3. The runtime controller contained elements to carry out two functions: transmitting the MQTT messages and controlling the timing. The runtime controller itself limited the total runtime of each trial, and used the $duration$ flag parameter to set the duration in seconds for each trial.

3.2.6 MQTT Pub Sampler

The MQTT pub (publish) sampler, shown in Figure 3.4, transmitted the MQTT messages to the pre-defined topic for each simulated client. The payload was a string in JSON format. The JSON fields in the MQTT pub sampler payload are enumerated in Table 3.5. Configuration parameters for the sampler itself are detailed in Table 3.6. The __time() method was used to retrieve the UNIX time stamp at tranmission and embed it in the message, which was parsed later to measure latency through the entire system.

3.2.7 Gaussian Random Timer

A guassian random timer element was configured to introduce noise present in the data sample . The sample standard deviation and mean were configured in this element. The

Table 3.5: MQTT Pub Sampler JSON fields and Values

JSON Field	JSON Value	Detail
timeTransmitted	`${__time()}`	transmission timestamp
thread	`${__threadNum}`	thread id within thread group
numClients	`${__P(clients,1)}`	clients parameter
trialDuration	`${__P(duration,1)}`	duration (seconds) parameter

Table 3.6: MQTT Pub Sampler Configuration Parameters

Setting	Value	Input Type
Name	<arbitrary >	text
Comments	<arbitrary >	text
Quality of Service	0	dropdown
retained messages	false	text
topic name	<arbitrary >	text
add timestamp in payload	false	radio
payloads	String	dropdown
payload	<see Table 3.5 >	text

$ConstantDelayOffset$ property was set to the mean, and the $Deviation$ property was set to the sample standard deviation.

3.3 MQTT and Database Ingest Pipelines

Data were transmitted and relayed using IoT Core integrated actions to write data to a MySQL Aurora cluster via a Lambda Function in one set of trials, and a DynamoDB table in a second set of trials. The DynamoDB table was set to enumerated values Write Capacity Units with Autoscaling enabled. The MySQL Aurora cluster consisted of a single db.r5.large instance. The MySQL Aurora cluster was configured as a single-master cluster with no read replicas.

3.3.1 MySQL Aurora Database

The MySQL Aurora Database was one of the two data destinations for MQTT messages ingested via the AWS IoT Core. Since at this time there is not a direct integration with

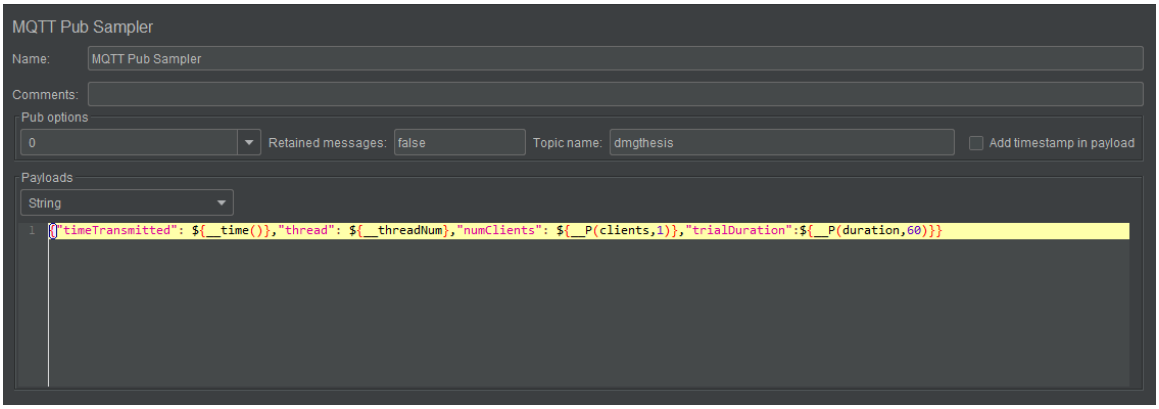

Figure 3.4: MQTT Pub Sampler

Table 3.7: Manufacturing Data Set Messages Schema

Column	Type	Note
messageId	int(11)	autoincremented primary key
dateInsert	datetime(3)	
payload	varchar(150)	JSON payload as string

MySQL Aurora in the IoT Core Actions that were triggered when MQTT messages were recieved, a Lambda Function rule action was triggered that utilized a shared pool of MySQL Connections. The Lambda Function was invoked using the IoT Core Rule Action, including the plain text MQTT message, which was parsed into a JSON object made up of key value pairs. The Lambda function code is provided in section A.1.

The database was initialized with a schema detailed in Table 3.7 that included a single table using the InnoDB engine. The Aurora Cluster was comprised of a single $db.r5.large$ instance function as the reader and writer.

3.3.2 DynamoDB Database

The DynamoDB Database was the second of the two potential data destinations for the MQTT messages ingested via the AWS IoT Core. DynamoDB features a direct integration with IoT Core, so the DynamoDB write IoT Core Rule Action was used to relay information. The DynamoDB table was created with a primary key of the Asset ID and a sort key

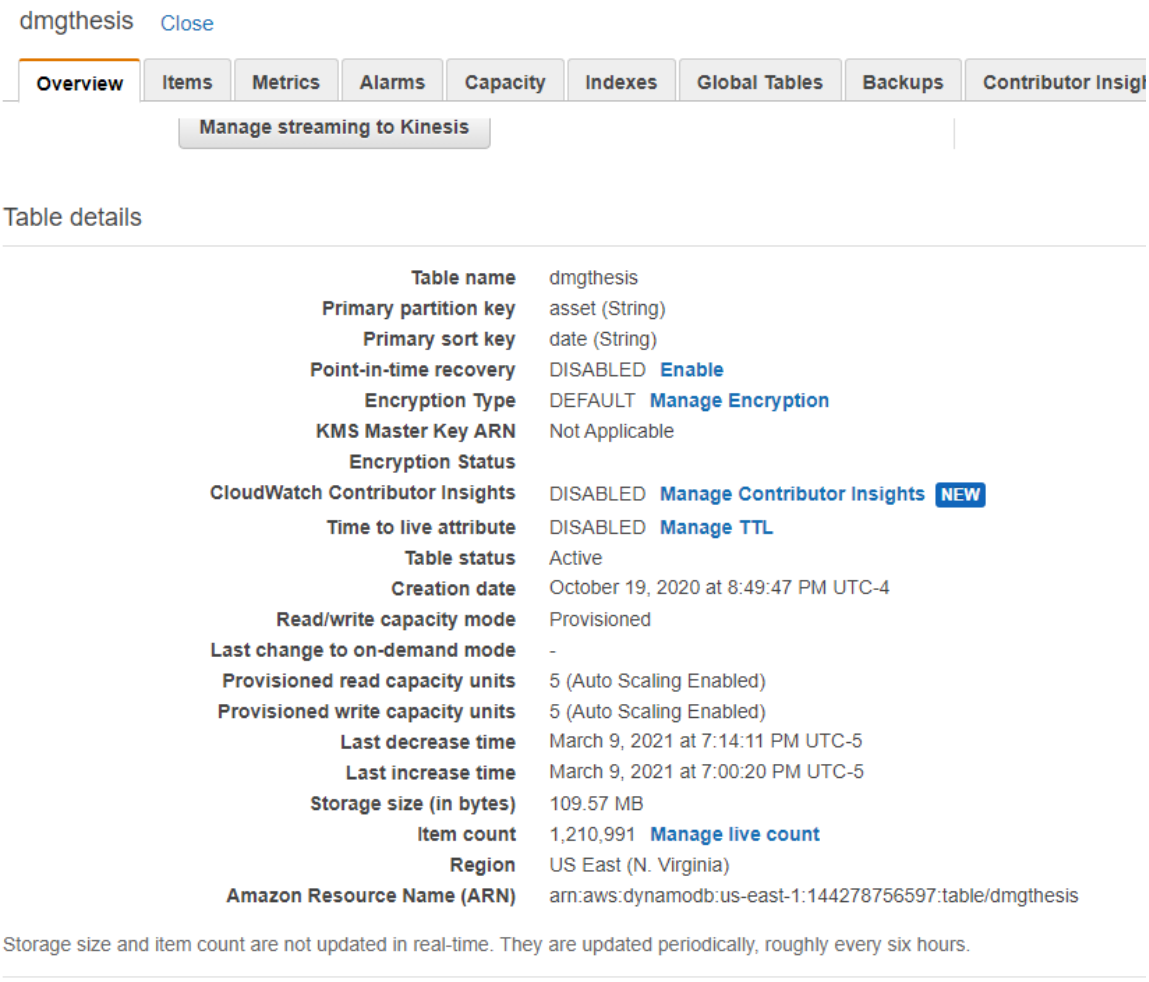

Figure 3.5: DynamoDB Table Configuration

comprised of the Unix Timestamp in Milliseconds of the message transmission time with configuration settings shown in Figure 3.5.

3.4 Remote Experiment Execution

The experiments were executed via Secure Shell (SSH) on the Elastic Compute Cloud (EC2) instance. Secure Copy Protocol (SCP) was used to copy the JMeter test plan files and execution scripts to the EC2 instance. Execution permission was added to the scripts with the following command:

```
chmod +x [bash script]
```

3.5 Latency Measurement

Latency metrics for both the NoSQL and MySQL systems were retrieved using the AWS python api with scripts included in section B.3. The trial start and end times were recorded from the command line via SSH for the execution of the load tests, and then entered into the decreasing_start and decreasing_end configuration variables. Since the architecture includes a lambda execution for the MySQL insertion, but the DynamoDB integration is fully managed, the full latency for the MySQL configuration is measured from the point of rule invocation onward. For MySQL this is defined by the Lambda execution duration, as the MySQL Insert operation itself occurs as the final code execution within this duration, and it includes the invocation delay that increases overall insertion latency.

3.6 Summary

The methods detailed in this chapters have outlined the procedures used for creating a full synthetic load-testing architecture that allows easy data collection via Cloudwatch Metrics. The proposed testing system was configured with both NoSQL DynamoDB and MySQL Aurora databases. The decoupled architecture used allows both databases to be connected in parallel for simultaneous evaluation. The ability to seed the simulation with parameters extracted from authentic manufacturing data allows characterization of system performance in terms of latency and throughput and validation of projected performance based on technology bottlenecks.

CHAPTER 4
RESULTS AND DISCUSSION

The results of the proposed synthetic load testing methodology are presented and evaluated in this chapter. The decoupled digital architecture is analyzed at the database insertion stage for both the NoSQL and MySQL configurations. Also, performance is evaluated in terms of database write throughput and insertion latency across database type and volume of connected simulated clients.

First, the end-to-end characteristics of the proposed synthetic load testing is evaluated for convergence to ensure the data throughput is stable. The stable experiment duration is determined across all proposed client test load sizes to isolate the effect of ramp-up in the time series data.

Next, the write performance of both isolated DBMS configurations are evaluated to establish a baseline performance for later benchmarking. Of the 57 assets included in the data set, 33 remain after excluding the testing/non-production assets. The size in bytes of the MQTT payload for the messages from valid assets is calculated. The mean and standard deviation of this signal is used for the simulated test assets.

Next, the results of request throttling via under-allocation of the $writecapacityunit$ parameter are presented. These results include consumed write capacity units, throttled request rate, and rate of client message receiving as recorded by AWS IoT Core Publish In Successes.

Next, the results of the database insertion latency for both NoSQL (DynamoDB) and MySQL (Aurora) are presented. The results are shown for both increasing and decreasing client load in order to account for auto-scaling momentum in which the DynamoDB throughput could be distorted. Verifying results with both increasing and decreasing client load configurations also accounts for Lambda container re-use that can drastically impact

latency via cold-start times. A summary of these results is provided at the end of the chapter.

4.1 End-to-End Characterization

Figures 4.1 to 4.5 show the end-to-end latency of the system from message transmission to message write completion on the Y axis in milliseconds, plotted over a range of trial durations in seconds on the X axis. Error bars reflect one sample standard deviation. The main objective of the end-to-end analysis is to evaluate the viability of trial durations for later experimentation. One objective is to identify trial durations that are too short, as they experience distortion of the latency by client initialization delays, which are not present in the system at a steady-state. The rapid decline and stabilization with increasing trial duration shown in figs. 4.1, 4.2 and 4.4 indicate that trials converge on the order of 10^1 to 10^2 seconds of trial duration. Figure 4.6 fails to converge, indicating that the 500 clients results are not indicative of a steady-state.

4.2 DynamoDB Writing

After evaluating the end-to-end system with a MySQL configuration, an isolated database verification for auto-scaling and request throttling was executed. Figure 4.7 shows DynamoDB consumed write capacity units, DynamoDB throttled requests, and IoT Core Successful publishes over MQTT for two trials with DynamoDB configured with 5 and 200 write capacity units, shown on the left and right halves, respectively. The simulated client load was 100 clients seeded with statistic parameters from the data set. The throttled requests were exclusively write requests to the DynamoDB table, since no read requests were executed during this time frame. The auto scaling burst capability of DynamoDB is responsible for the large spike in consumed write capacity units at 19:35 for the 5 write unit trial. The write requests were executed with a latency of less than 25ms for all messages. With auto-scaling enabled, the 200 write unit capacity was able to service the full load without

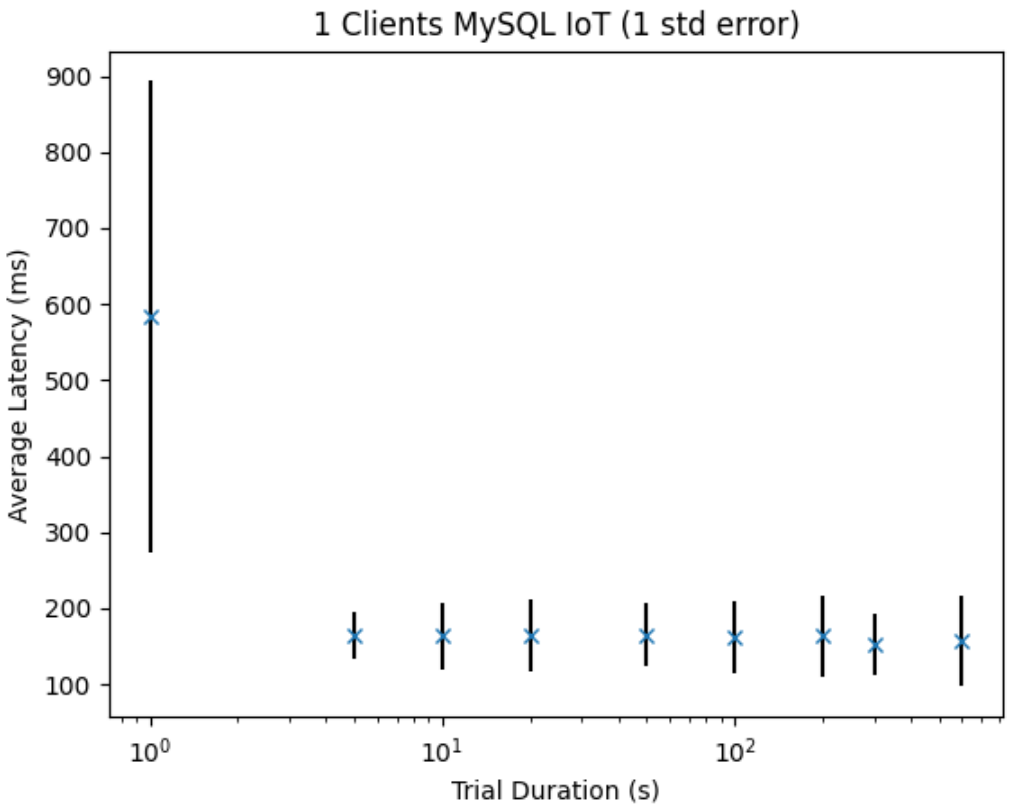

Figure 4.1: MySQL Average Latency with 1 Client, Variable Duration

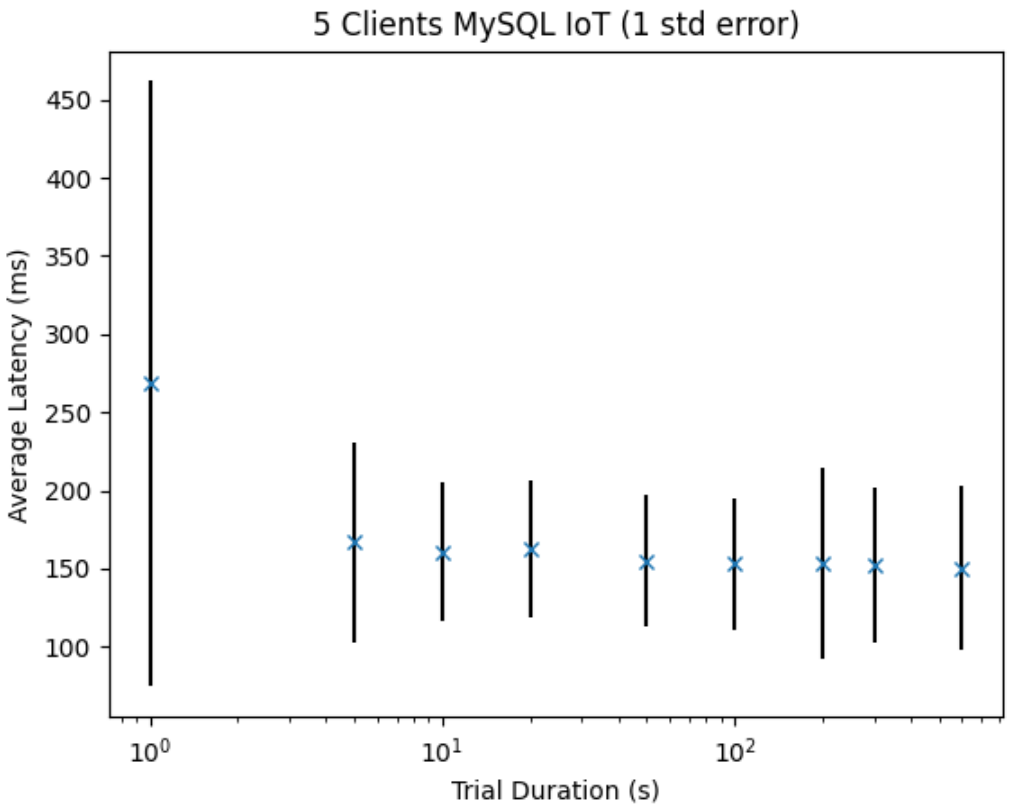

Figure 4.2: MySQL Average Latency with 5 Clients, Variable Duration

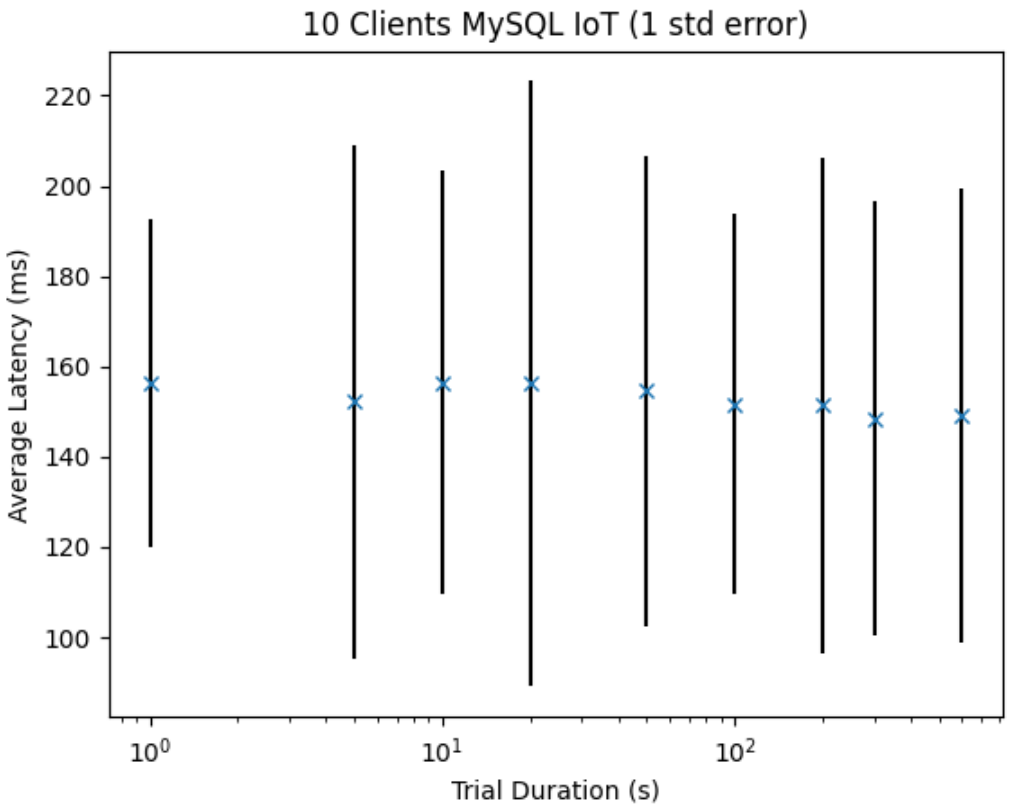

Figure 4.3: MySQL Average Latency with 10 Clients, Variable Duration

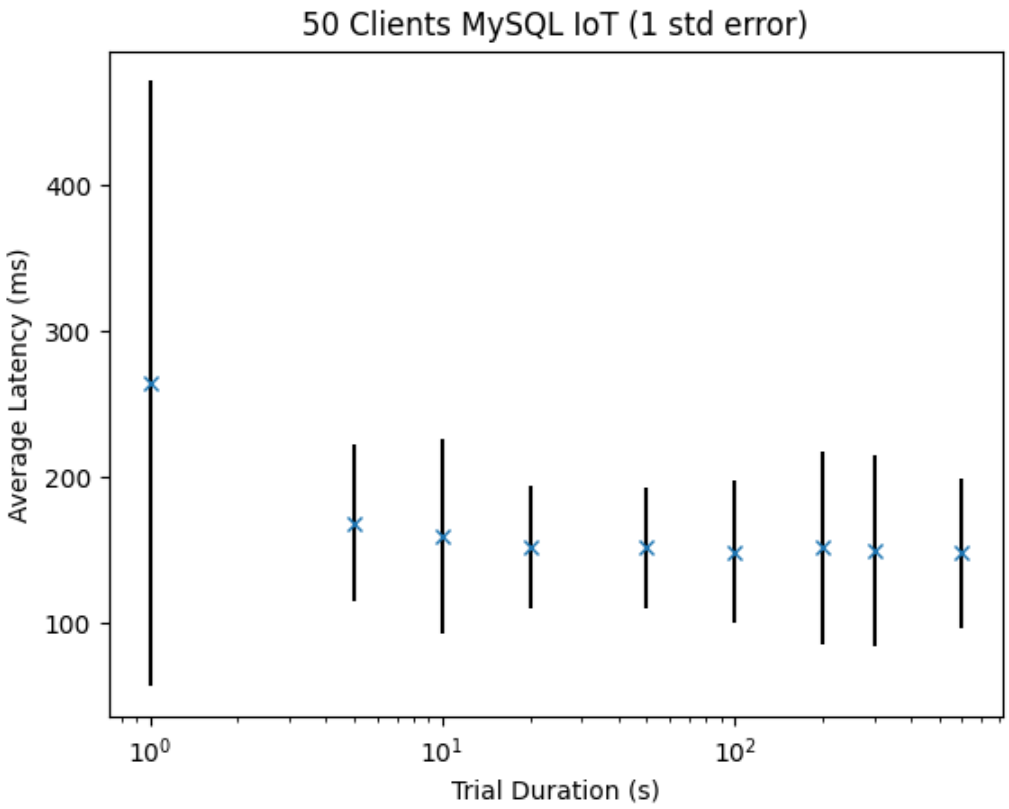

Figure 4.4: MySQL Average Latency with 50 Clients, Variable Duration

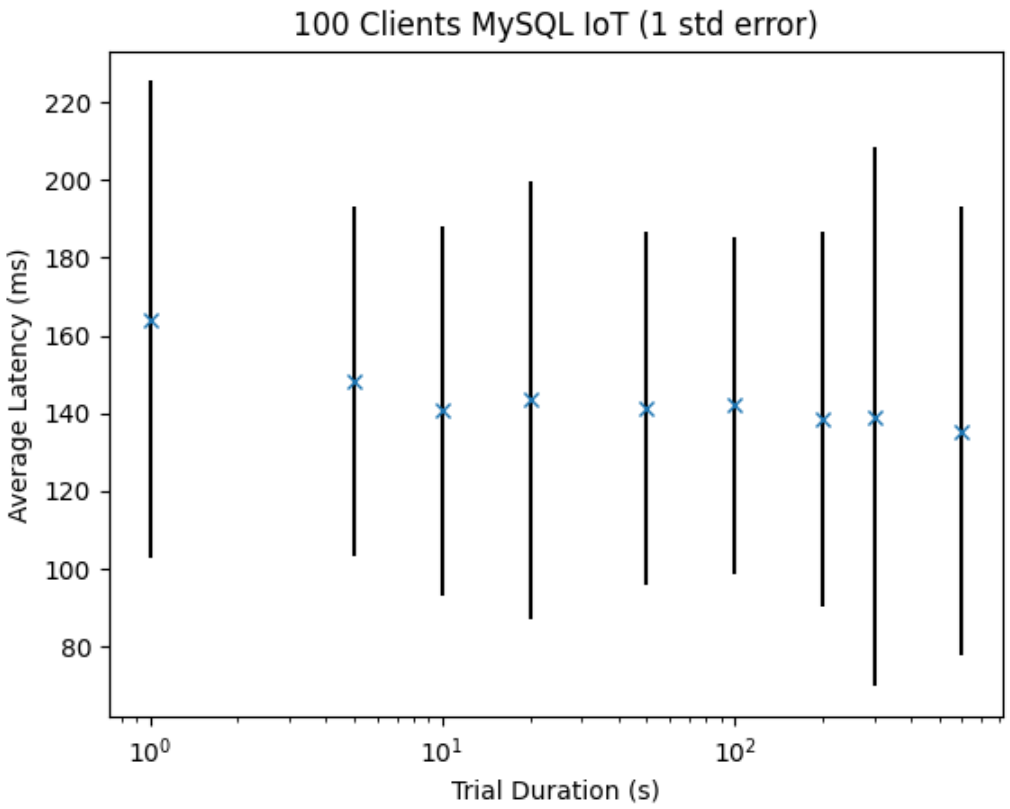

Figure 4.5: MySQL Average Latency with 100 Clients, Variable Duration

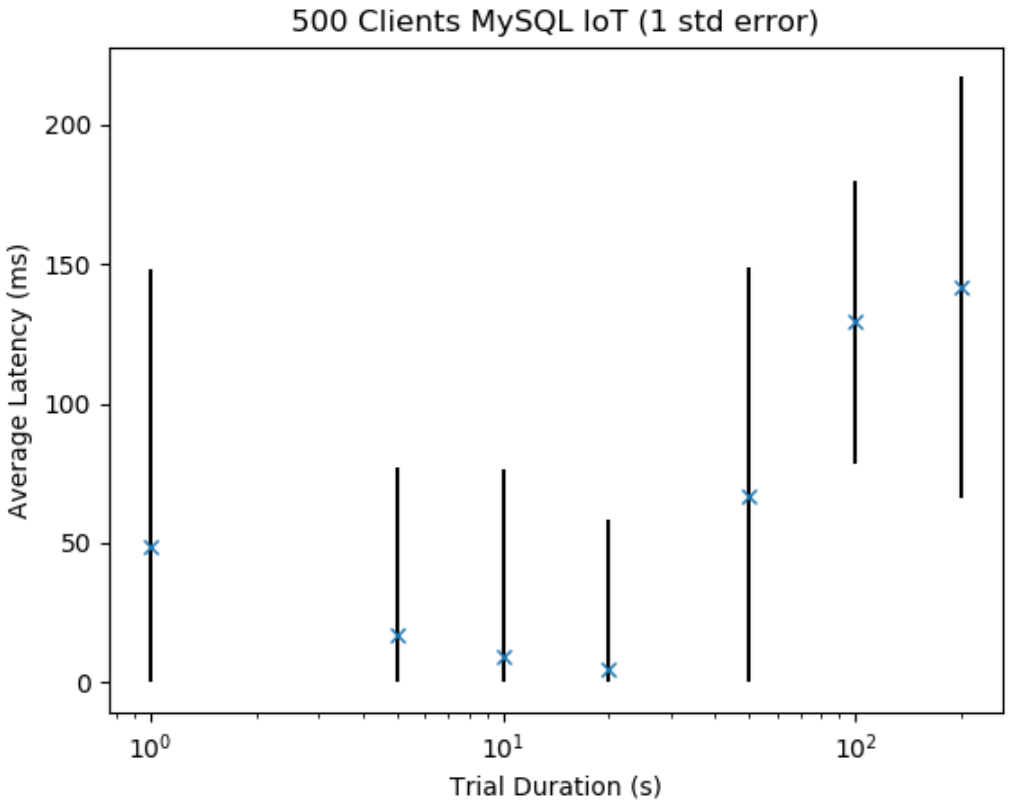

Figure 4.6: MySQL Average Latency with 500 Clients, Variable Duration

throttling any requests, while the 5 write units were unable to process the requests and bottlenecked into a large number of throttled requests.

4.3 MySQL Aurora Writing

The same test was executed using the MySQL Relational Database Management System (RDBMS) and observed to meet a write request latency of below 25ms as shown in Figure 4.8. These results are consistent with both existing literature and expected performance benchmarks. The MySQL trials used the same simulated 100 client configuration as the NoSQL trial. Throttling conditions were detected via the concurrent Lambda limits, since a direct integration with the AWS RDS service for writing records was not available, and Lambda was used to write messages from IoT Core to the RDS Instance. Through vertical scaling via increasing configured instance size, the RDS Instance can reach 200,000 writes per second, while DynamoDB by default is limited to 10,000 writes per second per table. The DynamoDB limit can be raised easily to far exceed RDS write limits, but at the cost of losing support for stream-enabled analytics.

4.4 NoSQL and MySQL Load Testing

Given the results of the two previous sections, the write throughput was serviced fully, resulting in identical throughputs for all configurations from NoSQL and MySQL; however, the latency differed in a statistically significant way. Figure 4.9 shows the latency of the isolated database portion of both configurations. For the DynamoDB configuration, the insertion is tracked through the managed monitoring solution integrated in AWS CloudWatch. For the MySQL configuration, the latency is primarily impacted by the Lambda function's invocation and execution time. The MySQL database Insert operation duration is included in this metric, as the function does not complete until the Insert operation is complete. An increase in rate of execution of Lambda functions can trigger new container provisioning, introducing cold-start delays. Executing the largest number of clients first

and then maintaining a monotonic, decreasing number of client connections concentrates the cold-start times in the time before the first trial. Trials with increasing client loads distribute this cold-start latency throughout the trials instead of aggregating most of the delay at the start of the experiment.

To ensure that the effects observed were not driven by Lambda cold-start behavior and DynamoDB auto-scaling momentum, the experiment was also run in reverse, and the results are shown in fig. 4.10. Both the increasing and decreasing load configurations show that the DynamoDB latency begins significantly higher and decreases with an intersection with Lambda insertion latency on the order of 10^1 client connections. The MySQL configuration that used Lambda functions to insert data did not change significantly with respect to the number of clients within the tested range, while the DynamoDB Insert Latency had an inverse relationship with the number of connected clients. Metrics were retrieved from AWS CloudWatch via the Python API using the scripts included the appendix section B.3.

4.5 Latency as an Architectural Factor

While the performance measurements in this work found an intersection of latency tradeoff between DynamoDB NoSQL and Aurora MySQL, several additional factors are practical in the design of an IoTfM or IIoT system that can have greater impact. By isolating the differences in latency due to choice of database as a relatively small component of the overall latency of an MQTT message's path, the ability to make architectural choices of database technology can be more heavily influenced by other factors, such as price, query flexibility, and ease of integration.

4.6 Scaling Prototype Systems

The results support the need for client load testing in the prototyping phase, as the latency and throughput respond differently to chosen database technologies as the number of clients increases from the range of 10 to 100. While many systems are tested for a

proof of concept using only a small number of instrumented assets, simulated testing with larger numbers of clients can reveal optimizations that would otherwise not be noticed until dozens of machines were already connected to an implemented system. Using data from already instrumented assets, the results show that the response of database write latency to increasing scale can be mapped prior to on-boarding larger numbers of assets. The ability to make informed decisions for architecture with respect to latency is especially important for warning systems, where latency can be critical. For small numbers of test clients, warnings would be received quickly, but as the number of clients increases, the choice of optimal database system becomes more complex.

4.7 Potential Flexibility and Advantages of a Decoupled Architecture

The decoupled architecture used in this work was found to provide effective interoperability between both the DynamoDB and RDS MySQL databases and the IoT traffic ingestion over MQTT. While an integrated IoT hub rule was used to write to DynamoDB, the initial traffic ingestion occurred for both test bench configurations over MQTT. The use of MQTT between loosely coupled components in the data processing pipeline could enhance the applicability of this work across different domains, as MQTT messages are extremely lightweight and can be sent from a variety of sources including other cloud providers, dedicated servers, and embedded devices. The ability to easily swap components within the architecture made isolating the database component significantly more accessible when compared to bespoke, tightly coupled pipelines.

For installations that adopt a decoupled, standardized architecture, the ability to quickly redirect IoT traffic to new destinations could enable faster upgrades and access to a wider range of technologies. Additionally, by using a standardized communication protocol between stages in data processing, multi-cloud configurations could be significantly easier to deploy. Since different cloud providers adjust pricing and deploy new features independently, users of the decoupled architecture could stand to gain more quickly from advances

on any cloud provider with potentially drastic reductions in cost of adoption.

4.8 Summary

The results illustrate comparable performance in terms of latency across the NoSQL and MySQL implementations. As a decoupled architecture was used, the full end-to-end system's latency was found to converge for simulated client loads less than 500 clients, and the convergence was found to occur at these load parameters at trial durations greater than 100 seconds. The individual performance results from DynamoDB NoSQL confirmed write unit capacity expectations for provisioning throughput, and auto-scaling and burst behaviors were observed to exhibit limited momentum in provisioned throughput. The MySQL Aurora isolated insertion testing was bottlenecked by Lambda executions, as the requirement to trigger Lambda functions from IoT Core rules added an order of magnitude of latency to the system. MySQL Aurora insertions occurred with latency of less than 0.25 milliseconds, while the Lambda invocation and execution introduced greater than 4.0 milliseconds of latency. Both increasing and decreasing client load execution orders of simulated client loading trials indicated that the DynamoDB insertion latency began higher than the Lambda, but decreased as client load increased. After an intersection between 10^1 and 10^2 simulated clients, the Lambda configuration maintained its latency, while DynamoDB performed with approximately 50% less latency.

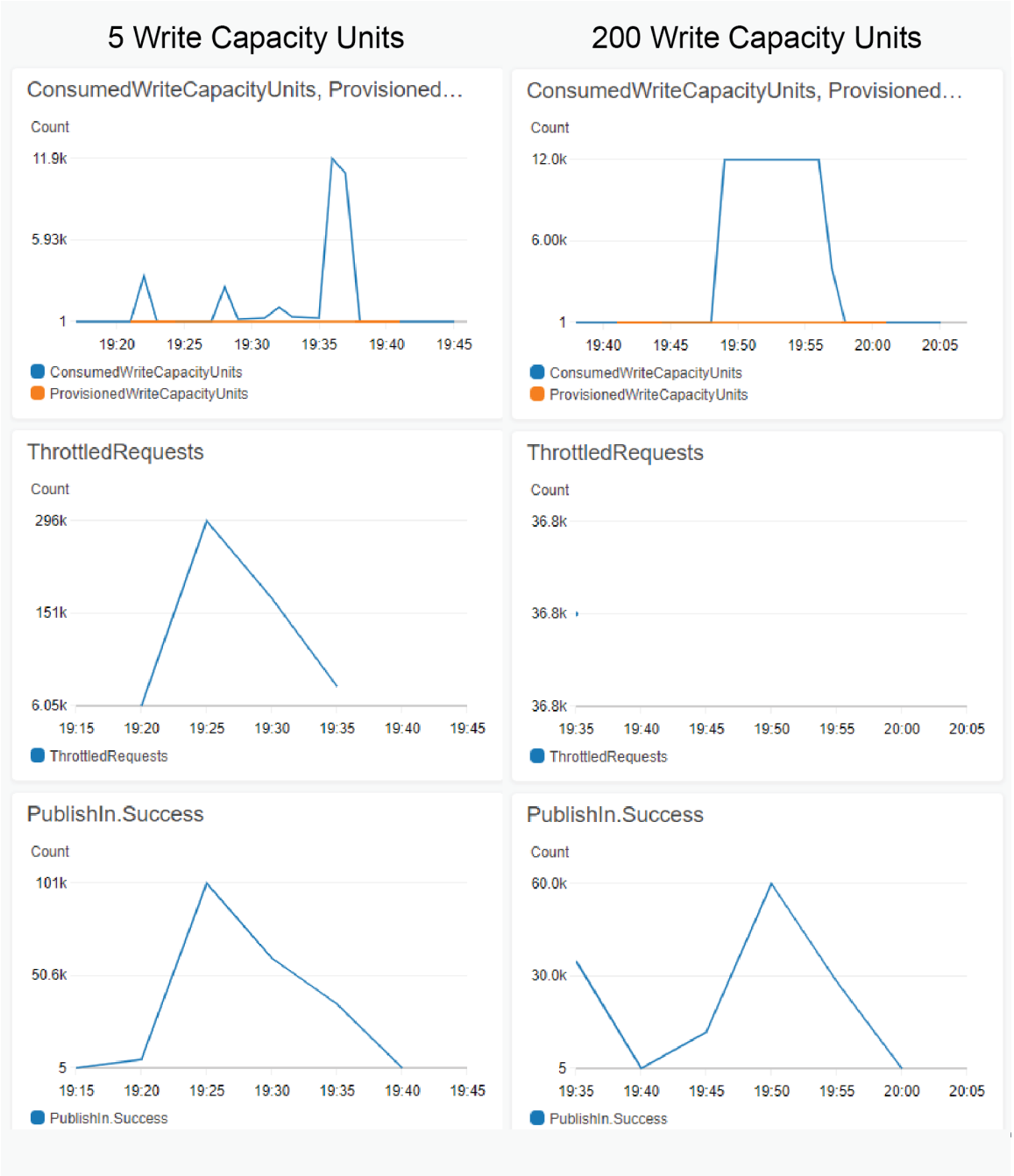

Figure 4.7: DynamoDB 100 Clients Trial

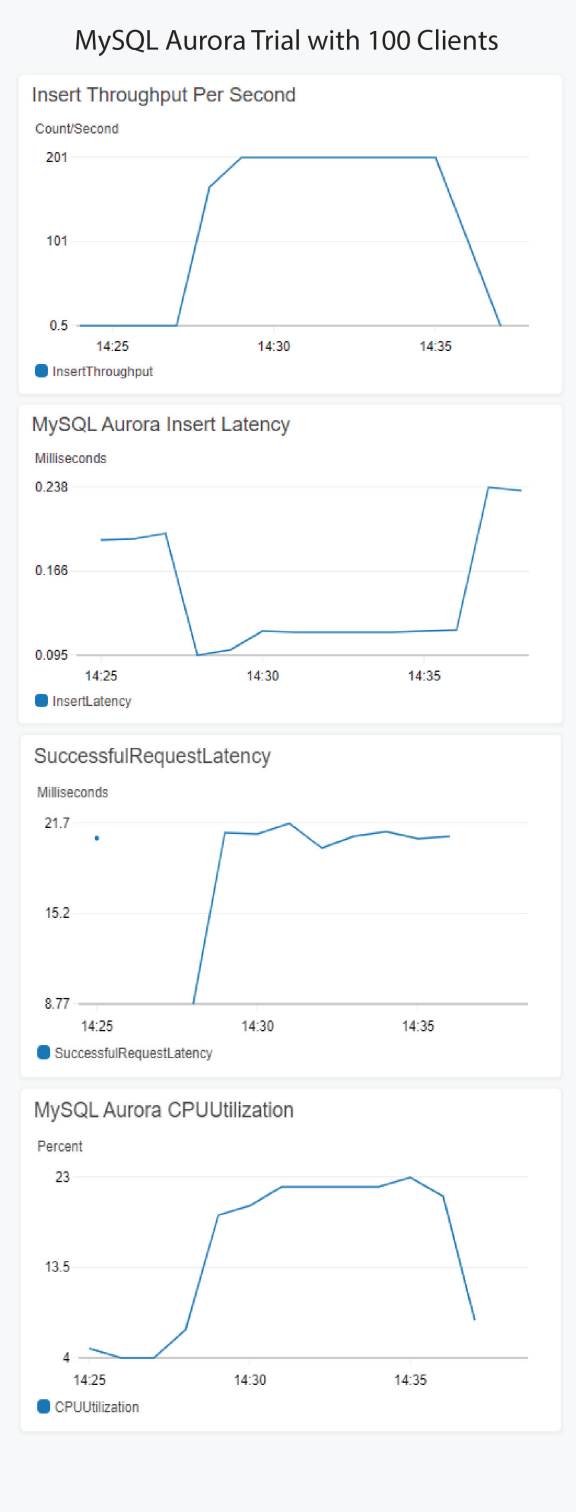

Figure 4.8: Aurora MySQL 100 Clients Trial

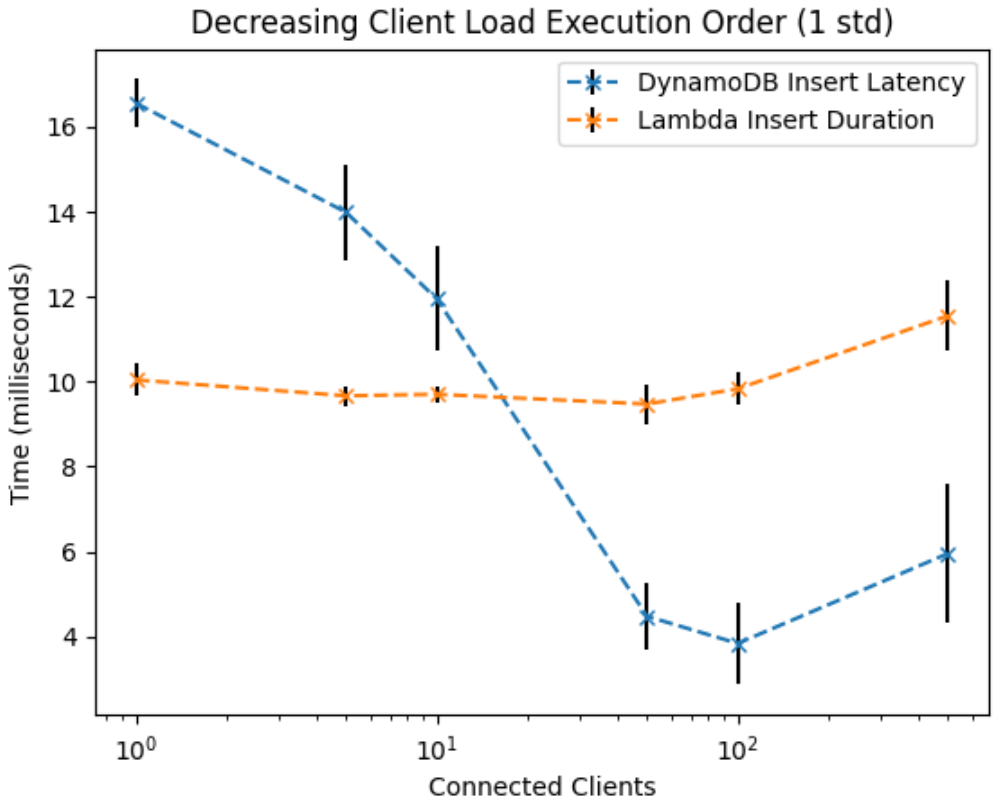

Figure 4.9: MySQL and NoSQL Isolated Latency - Decreasing Load

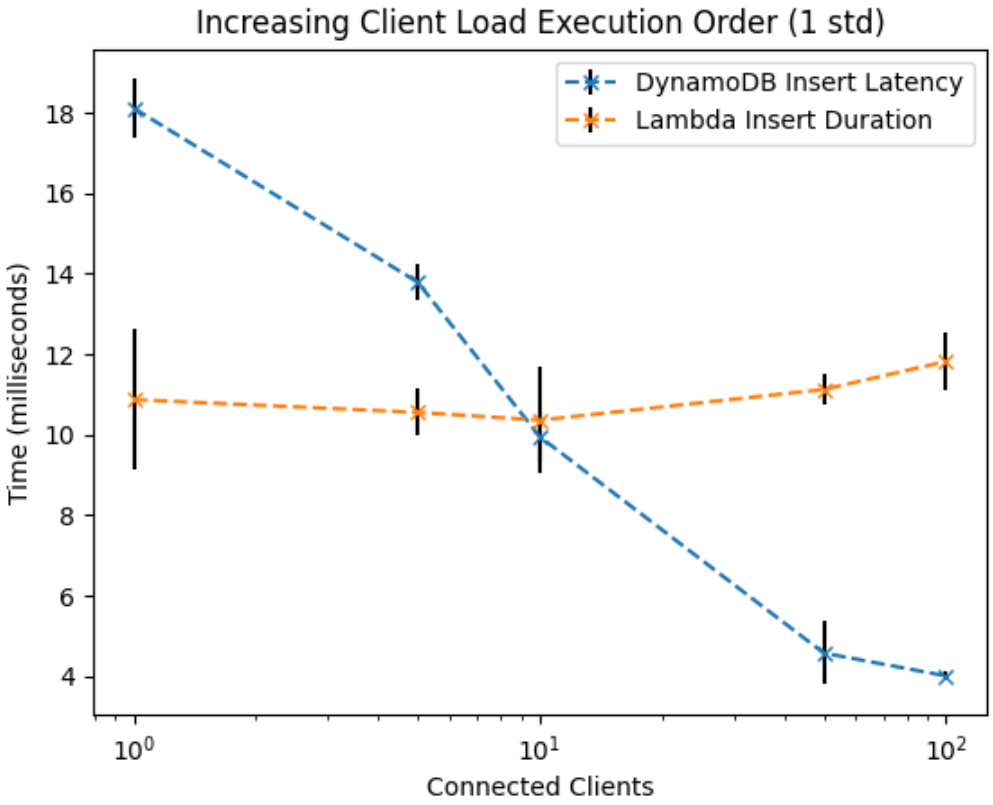

Figure 4.10: MySQL and NoSQL Isolated Latency - Increasing Load

CHAPTER 5
CONTRIBUTIONS

The contributions of this work are detailed below.

5.1 Simulated Client Testing for a Decoupled Architecture

A method for testing a decoupled architecture using simulated clients is provided in this work. The simulated clients testing reveals bottlenecks that are not inherent to any individual layer within the architecture, and the results show that it can be used for bench-marking vastly different database technologies using a common set of metrics, such as latency and throughput.

5.2 Simulated Client Generation from Historic IoTfM Records

The process for extracting and generating simulated clients that generate randomized traffic while maintaining mean characteristics from a pre-defined backlog of IoT records is demonstrated to provide a testing environment simulating production work loads compared to directly stress-testing the database layer itself.

5.3 Isolation of Database Performance within Decoupled Architecture

A method for measuring database latency and throughput metrics from a decoupled architecture was detailed in this work. The method leverages Cloudwatch metrics to allow high-resolution analysis of performance by excluding latency from the MQTT messaging layer. The methods introduced could provide direction for instrumenting similar decoupled architectures in other cloud provider environments or non-cloud installations. Results support that the metrics gathered are able to provide non-obvious insights into performance trade-

offs within the database layer of the decoupled architecture. Isolating the latency within the database layer could be used to inform architectural decisions and evaluate potential alternative database technologies for future research on IoTfM system configuration.

CHAPTER 6
FUTURE RESEARCH

The work presented here can be expanded in several directions to further enhance and explore the field of IoTfM. Characterizing the impact of instance resources such as Random Access Memory (RAM) and Central Processing Unit (CPU) core count could provide greater insights into vertical scaling capabilities of these decoupled architectures, revealing optimal points to relieve bottlenecks. This work was limited to only two kinds of databases, both of which were managed AWS offerings. Broadening this work to include both other cloud providers and a greater variety of database technologies would lead to a more holistic picture of databases in IoTfM. Applying the same sampling and simulation methods from this work to other manufacturing firms' historical data sets could provide insights into different industrial data gathering practices that would greatly contribute to the ability to generalize the performance data acquired through these experiments.

6.1 Query Flexibility

As in the practical experiment conducted and described above, the choice of database architecture is not differentiated most significantly by performance when applied to the Industrial IoT. The ability to flexibly query data is a feature that is far more developed and advanced on RDBMS systems like MySQL. Flexible queries can be instrumented on top of NoSQL databases via Extract Transform Load (ETL) pipelines or distributed Big Data approaches, such as Hadoop. For a streamlined architecture in line with literature, the number of discrete data storage locations are minimized, and systems organically grow from proof of concept and pilot installations. In these scenarios, MySQL's combination of established high vertical scaling write speed limit and facilitation of higher-level analytics beyond simple data archiving and arithmetic-based stream metrics allows for faster

realization of higher level CPPS systems.

6.2 Latency Sensitivity

Latency from the JMeter EC2 to the IoT Core was negligible throughout the experiments, since the JMeter instance was in the same AWS region as the IoT Core endpoint, and it was connecting within AWS instead of being transmitted from a manufacturing location. This physical co-location is what made it possible to execute the experiments without the introduction of latency noise from outside communications and network traffic that could potentially have vastly greater influence on the metrics measures than the ones investigated in this work. Network and internet service provider offerings could influence the maximum viable throughput from a CPPS when transmitting payloads to the cloud, as would any bottlenecks in factory floor networking. In many manufacturing scenarios, wireless communication over wifi or bluetooth is used as well, which introduces another order of magnitude of latency variability. Examining the system-level effects of additional sources of latency such as shared network resources and environment-wide trends in network congestion is another looming problem in need of precise characterization when narrowed to the IoTfM field.

6.3 Architectural Scaling

The high-fidelity simulated industrial MQTT sensor payloads and publish characteristics validate a more specialized testing model to potentially allow higher resolution data on future results compared to generalized DBMS throughput and latency comparisons. Due to the MQTT pub/sub architecture, even for scenarios in which a single MySQL instance would saturate its write capacity, MySQL could be used by splitting data from different assets into separate databases or leveraging multi-master clusters. By horizontally scaling the IoT architecture itself using multiple databases subscribed to distinct topics instead of scaling the database management system, CPPS data throughput bottlenecks can be fully

avoided. Fitting a multi-database system in which databases are subscribed to topics over MQTT could benefit from leveraging read-replicas to allow analytics to run on separate replicas of the databases without impacting the CPU load of the write instance, maximizing theoretical write throughput.

CHAPTER 7

LIMITATIONS

7.1 Instance Types

Since results were derived using particular instance sizes and configurations, the results have limited predictive power when other sized instances are used. Memory and CPU limitations on the EC2 instance used to execute the JMeter tests was prohibitive in that it limited tests to below 500 connected clients. While the work aimed to initially explore larger numbers of clients, the simulation framework itself experienced throughput limitations that necessarily capped the maximum number of clients. Differently optimized instances could present a different set of tradeoffs, such as higher network connectivity speeds or more optimizations towards high thread count computation and parallel processing.

7.2 Cloud and Database Technologies

Given the vast array of potential database storage technologies, configurations, and hosted services, the work presented here characterizes a particular use case through AWS integrated offerings. Other cloud providers offer different technology stacks and integrations that fundamentally alter both throughput and latency results. Only one platform, AWS, was used to generate the results in this work. Other Cloud providers could implement the same technologies in different ways, or allow different levels of user control granularity for configuration parameters that were used in this work. Integrated monitoring and metrics interfaces are implemented differently across providers, and methods for gathering the same results outside AWS are not investigated in this work.

7.3 Manufacturing IoT Data Set

The data set used for characterizing manufacturing data used for this work is not a generalized IoTfM data set, and provides a granular, real-world example at the cost of broader scope. The data set is used to ensure fidelity with authentic manufacturing sensor readings and conditions. Manufacturing data varies in characteristics, format, and metadata across installations, industries, and environments. The results of this work are necessarily limited to the data that was examined within the manufacturing industry.

7.4 Cloud Services Offerings

In the emerging space of cloud services and hosted databases, services and their availability are regularly subject to change. Future versions of technology offerings can fundamentally alter functionality and add or remove features. Accounting for long-term trends in feature development is not examined in this work, and all technologies discussed are subject to change.

7.5 Cost and Pricing

Cost and pricing analysis is not included in this report, as it is rapidly evolving in the cloud and IoTaaS space. While theoretical throughput optimizations are promising and enticing, the cost analysis for implementing many of the systems and methods demonstrated in this work can vary by billing model and cloud provider. The sensitivity of private sector implementations to pricing and cost is not examined nor accounted for in this work.

7.6 Environmental Impact

The environmental impact of IoT databases and data ingestion pipelines is not examined in this work. Cloud technology makes it much easier to use a tremendous volume of computational resources, but also can result in more efficient reuse of servers as idle resources are

able to be repurposed by another user. As the volume of data from the IoT grows, the environmental cost of storing such data efficiently can be impacted by the format and system used to store the data. Higher-availability data systems tend to consume more electricity and have larger impacts on the environment. Longer-term storage can reduce responsiveness of data querying, but provide potentially dramatic energy savings.

CHAPTER 8

CONCLUSIONS

The ability to convert ingested data flexibly into higher-level insights via dynamic access patterns makes MySQL a strong fit for IoT for Manufacturing Applications using AWS. Direct write speed and latency at scale yield better performance over 200k message writes per database per second for NoSQL as compared to SQL, yet the impact could be fully alleviated by splitting data writing across multiple databases using a decoupled architecture with multiple write database instances or multi-master MySQL cluster configurations. The capability to derive complex, dynamic insights from SQL aligns best with Industry 4.0 objectives of smart manufacturing by allowing flexibly-defined access patterns, while NoSQL requires well-defined access patterns. Stream and direct storage recall without analytics implementations are better served by the scalability of NoSQL. NoSQL can facilitate lower level data storage but requires additional technologies to explore higher level insights, and NoSQL can require knowledge of necessary access patterns in advance.

8.1 Restatement of Hypotheses

The generalized database results from established work will hold true in the narrow use case of IoTfM installations in that NoSQL will experience significantly greater throughput and lesser latency as compared to MySQL.

8.2 Testing the Hypothesis

The end-to-end and database-isolated latency tests support the proposed hypothesis in that write latency was lower for the NoSQL DynamoDB configuration than for the MySQL Aurora configuration. The hypothesis was supported for larger client load sizes over 10^1,

but for smaller client loads the MySQL Aurora configuration had a higher latency than the NoSQL DynamoDB.

8.3 Restatement of Research Questions

This work aims to address these needs with the following research objectives:

1. characterizing the scalability behaviors of NoSQL and SQL databases in the context of existing CPS and CPPS frameworks
2. enumerating scaling factors and bottlenecks encountered during synthetic load tests seeded with data from a major US manufacturing firm

8.4 Answers to Research Questions

The results from this work can be applied to answer the research questions previously enumerated.

1. The scalability of both NoSQL and SQL databases examined in this work fall within the first two layers of the 5Cs model as described in Figure 2.4. The scalability of these systems is critical to enabling higher levels of the model to develop. Within a decoupled architecture, the ability to interchange databases allows for greater flexibility, and the work presented here allows evaluation of DBMS with respect to performance via latency and throughput analysis. The scalability of both NoSQL and SQL databases can be compared over increasing client load conditions using simulated clients to determine performance differences. With respect to enabling higher levels of the 5Cs CPS model, MySQL's ability to derive higher level insights and enforce data constraints can offer more towards analytics. NoSQL can be optimized for lower latency in use cases that don't rely on flexible access patterns.
2. Scaling bottlenecks were encountered both for the synthetic load testing system itself, and with the decoupled architecture model in both database configurations. For

the NoSQL DynamoDB configuration, the most prevalent bottleneck observed was the write capacity unit limitation, in which insufficient write capacity units were provisioned, and the throttled insertion requests rapidly grew. For the MySQL configuration, the Lambda function insertion stage was the primary bottleneck, as it introduced cold-starts to initialization of the system and with each each increase in load. The MySQL configuration also was subject to AWS account limits on maximum concurrent Lambda function invocations; however, this limit can be raised via support tickets. The load testing instance itself experienced a bottleneck in simulated client thread execution for the trials with 500 clients, which could be resolved with vertical scaling via a larger provisioned EC2 instance.

Appendices

APPENDIX A

EXPERIMENT CODE

The following code was executed in the specified environments in order to run the experiments.

A.1 MySQL Lambda Code

The following code was run in the Lambda function to pass data from IoT Core to MySQL Aurora

```
// Function to insert data into the Messages table.
var mysql = require('mysql');
var config = require('./config.json');

var pool  = mysql.createPool({
    host     : config.dbhost,
    user     : config.dbuser,
    password : config.dbpassword,
    database : config.dbname
});

exports.handler =  (event, context, callback) => {
    context.callbackWaitsForEmptyEventLoop = false;
    pool.getConnection(function(error, connection) {

        // Variable definitions
```

```
let fields = [];
let payload = "";
let resultsObject;
let sql = "";
let table = "messages";
let topic = "dmgthesis";
let values = [];

console.log(event);

// Place the payload and topic in the fields and
   values array.
payload = JSON.stringify(event);

// Insert the data into the database.
sql = 'INSERT INTO ' + table + ' (payload) VALUES ?'
   ;
connection.query(sql, [[[payload]]], function (error
   , results, fields){
    connection.release();
    if (error) { callback(error);}
    else {
        resultsObject = JSON.parse(JSON.stringify(
           results));
        if ( resultsObject.affectedRows == 1)
            callback(null, 'Success');
        else
```

```
                    callback(null, 'Failure');
                }
            });

    });
};
```

A.2 MySQL Aurora Duration Bash Script

```
#!/bin/bash
cd apache-jmeter-5.3
cd bin

## Config vars

#echo Begin trials with $EXPERIMENT_DURATION_SECONDS second
    duration
#echo
#1 5 10 20 50 100 200 300 600



for NUM_CLIENTS in 1 5 10 50 100 500
do
        for EXPERIMENT_DURATION_SECONDS in 1 5 10 20 50 100
            200 300 600
        do
                echo
                    ----------------------------------------


                echo Running trial with $NUM_CLIENTS clients
                    , $EXPERIMENT_DURATION_SECONDS seconds
                echo
                    ----------------------------------------
```

```
                echo
                ./jmeter -n -t ../iotCore_1thread.jmx -l ../
                    iotcore2.jtl -Jclients=$NUM_CLIENTS -
                    Jduration=$EXPERIMENT_DURATION_SECONDS
        done
done
```

APPENDIX B

DATA PROCESSING

The following code and queries were executed to extract, filter, and process the raw data. Visualizations were generated using included python scripts that use matplotlib to create plots.

B.1 MySQL Aurora Latency Query

```
SELECT numClients,trialDuration,AVG(latency),stddev(latency)
    ,COUNT(latency),min(latency),max(latency)
FROM(
        SELECT dInsert,dTransmit, dInsert*1000 - dTransmit
            latency, numClients, trialDuration from (
                SELECT
                messageId,
                dateInsert,
                payload,
                unix_timestamp(dateInsert) dInsert,
                JSON_EXTRACT(payload,'$.timeTransmitted')
                    dTransmit,
            JSON_EXTRACT(payload,'$.numClients') numClients,
                JSON_EXTRACT(payload,'$.trialDuration')
                    trialDuration
                FROM messages
                WHERE
```

```
                JSON_VALID(payload)
        ) AS messagesWithDates
) insertLatencies
WHERE latency < 10000
GROUP BY numClients, trialDuration;
```

B.2 MySQL Trial Duration Plot Script

```
import matplotlib.pyplot as plt
import pandas as pd

#sweep = pd.read_csv('3 10 2021 variable sweep.csv')
sweep = pd.read_csv('3 10 2021 variable sweep filtered 10k.
    csv')



client_nums = sweep['# numClients'].drop_duplicates()

for client_num in client_nums:
    plt.figure(client_num)
    # Rows within this sample that match our client num
    trial_rows = sweep[sweep['# numClients'] == client_num]

    print(trial_rows.columns)
    plt.xscale('log')

    plt.errorbar(trial_rows[' trialDuration'],
                 trial_rows[" AVG(latency)"], trial_rows["
                     stddev(latency)"],
                 ls='None',
                 marker='x',
                 ecolor='black')
```

```
        plt.xlabel('Trial Duration (s)')
        plt.ylabel('Average Latency (ms)')

        plt.title('{} Clients MySQL IoT (1 std error)'.format(
            client_num))
        plt.savefig(
            'sweep_plots/mysql_average_latency_{}_clients.png'.
                format(client_num))

print('done')
```

B.3 Fetch Latency Metrics Script

```
import boto3
from datetime import datetime
import pandas as pd

# Create CloudWatch client
cloudwatch = boto3.client('cloudwatch')

# Isolate Trial Execution times
decreasing_start = datetime.fromisoformat('2021-03-15T21
    :36:00')
decreasing_end = datetime.fromisoformat('2021-03-15T22:44:00
    ')

increasing_start = datetime.fromisoformat('2021-03-09T19
    :17:00')
increasing_end = datetime.fromisoformat('2021-03-09T20:28:00
    ')

clientNums = [1, 5, 10, 50, 100, 500]


def get_x_metrics(start, end):
    response = cloudwatch.get_metric_data(
        MetricDataQueries=[
            {
```

```
        'Id': 'ddbInsertLatency',
        'MetricStat': {
            'Metric': {
                'Namespace': 'AWS/DynamoDB',
                'MetricName': '
                    SuccessfulRequestLatency',
                'Dimensions': [
                    {
                        'Name': 'TableName',
                        'Value': 'dmgthesis'
                    },
                    {
                        'Name': 'Operation',
                        'Value': 'PutItem'
                    },
                ]
            },
            'Period': 60,
            'Stat': 'Average',
            'Unit': 'Milliseconds'
        },
        'Label': 'string',
        'ReturnData': True,
    },
    {
        'Id': 'lambdaDuration',
        'MetricStat': {
```

```
            'Metric': {
                'Namespace': 'AWS/Lambda',
                'MetricName': 'Duration',
                'Dimensions': [
                    {
                        'Name': 'FunctionName',
                        'Value': 'dmg-thesis-sql-
                            insert'
                    },
                ]
            },
            'Period': 60,
            'Stat': 'Average',
            'Unit': 'Milliseconds'
        },
        'Label': 'string',
        'ReturnData': True,
    },
    {
        'Id': 'rdsInsertLatency',
        'MetricStat': {
            'Metric': {
                'Namespace': 'AWS/RDS',
                'MetricName': 'InsertLatency',
                'Dimensions': [
                    {
                        'Name': '
```

```
                        DBInstanceIdentifier',
                    'Value': 'dmg-thesis-
                        instance-1'
                },
            ]
        },
        'Period': 60,
        'Stat': 'Average',
        'Unit': 'Milliseconds'
    },
    'Label': 'string',
    'ReturnData': True,
},
{
    'Id': 'iotCoreRuleSuccess',
    'MetricStat': {
        'Metric': {
            'Namespace': 'AWS/IoT',
            'MetricName': 'Success',
            'Dimensions': [
                {
                    'Name': 'ActionType',
                    'Value': 'DynamoDB'
                },
                {
                    'Name': 'RuleName',
                    'Value': 'dmgthesis_dynamo'
```

```
                        },
                    ]
                },
                'Period': 60,
                'Stat': 'SampleCount',
                'Unit': 'None'
            },
            'Label': 'string',
            'ReturnData': True,
        },
    ],
    StartTime=start,
    EndTime=end,
    ScanBy='TimestampAscending',
    MaxDatapoints=500,
    LabelOptions={
        'Timezone': '+0000'
    }
)

metricResults = response['MetricDataResults']

# print(response)
for result in metricResults:
    print('{} len={}'.format(result['Id'], len(result['
        Values'])))
```

```
# Extract Metrics and assert correct key ordering
assert(metricResults[0]['Id'] == 'ddbInsertLatency')
ddbInsertLatency = metricResults[0]['Values']

assert(metricResults[1]['Id'] == 'lambdaDuration')
lambdaDuration = metricResults[1]['Values']

assert(metricResults[2]['Id'] == 'rdsInsertLatency')
rdsInsertLatency = metricResults[2]['Values']

assert(metricResults[3]['Id'] == 'iotCoreRuleSuccess')
iotCoreRuleSuccess = metricResults[3]['Values']

# Convert to dictionary for pandas Dataframe constructor
decreasing_load_dict = {
    'ddbInsertLatency': ddbInsertLatency,
    'lambdaDuration': lambdaDuration,
    'rdsInsertLatency': rdsInsertLatency,
    'iotCoreRuleSuccess': iotCoreRuleSuccess
}

# Convert to dataframe
df = pd.DataFrame(decreasing_load_dict)

#
df['Clients'] = df['iotCoreRuleSuccess'] / 120.0
```

```
    def round2nearest_client_num(val):
        return min(
            clientNums,
            key=lambda x: abs(x-val)
        )

    # Round to the closest number of clients instead of
      estimated
    df['Clients'] = df['Clients'].apply(
      round2nearest_client_num)
    return df


increasing_df = get_x_metrics(increasing_start,
  increasing_end)
decreasing_df = get_x_metrics(decreasing_start,
  decreasing_end)

increasing_df.to_csv('increasing1.csv')
decreasing_df.to_csv('decreasing1.csv')

print(increasing_df)
print(decreasing_df)
```

REFERENCES

[1] H. Kagermann and W. Wahlster, *Industrie 4.0 - germany market report and outlook*, 2016.

[2] V. Nguyen and A. Dugenske, "An internet of things for manufacturing (iotfm) enterprise software architecture," *Smart and Sustainable Manufacturing Systems*, vol. 2, no. 2, pp. 177–189, 2018.

[3] D. Mourtzis, E. Vlachou, and N. Milas, "Industrial big data as a result of iot adoption in manufacturing," *5th CIRP Global Web Conference Research and Innovation for Future Production*, 2016.

[4] N. Briscoe, "Understanding the osi 7-layer model," *PC Network Advisor*, vol. 120, no. 2, 2000.

[5] T. G. Handel and M. Sandford, "Hiding data in the osi network model," *International Workshop on Information Hiding*, pp. 23–38, 1996.

[6] J. Postel, "Internet protocol," 1981.

[7] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč, "Foundations of json schema," in *Proceedings of the 25th International Conference on World Wide Web*, 2016, pp. 263–273.

[8] P. V. Biron, A. Malhotra, W. W. W. Consortium, *et al.*, *Xml schema part 2: Datatypes*, 2004.

[9] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, *Hypertext transfer protocol–http/1.1*, 1999.

[10] M. Parto Dezfouli, "Automated real-time machine learning for iot for manufacturing a cloud architecture and api," Ph.D. dissertation, Georgia Institute of Technology, 2019.

[11] J. Postel, *Rfc0768: User datagram protocol*, 1980.

[12] ——, *Rfc0791: Transmission control protocol rfc 793*, 1981.

[13] D. T. R. Monitoring, *What's the default snmp port number? is snmp tcp or udp?* https://www.dpstele.com/snmp/transport-requirements-udp-tcp.php, 2020.

[14] *Mqtt: The standard for iot messaging*, 2020.

[15] D. Soni and A. Makwana, "A survey on mqtt: A protocol of internet of things (iot)," in *International Conference On Telecommunication, Power Analysis And Computing Techniques (ICTPACT-2017)*, vol. 20, 2017.

[16] T. Yokotani and Y. Sasaki, "Comparison with http and mqtt on required network resources for iot," in *2016 international conference on control, electronics, renewable energy and communications (ICCEREC)*, IEEE, 2016, pp. 1–6.

[17] Z. Shelby, K. Hartke, and C. Bormann, "The constrained application protocol (coap)," 2014.

[18] S. Vinoski, "Advanced message queuing protocol," *IEEE Internet Computing*, vol. 10, no. 6, pp. 87–89, 2006.

[19] N. Naik, "Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http," in *2017 IEEE international systems engineering symposium (ISSE)*, IEEE, 2017, pp. 1–7.

[20] B. Hayes, "Cloud computing," *Communications of the ACM*, vol. 51, no. 7, pp. 9–11, 2008.

[21] P. Mell and T. Grance, "The nist definition of clouding computing recommendations national inst. of standards and technology," *NIST Special Publication*, vol. 145, p. 7, 2011.

[22] M. Miller, "Cloud computing: Web-based applications that change the way you work and collaborate online," *Indiana: QUE*, 2008.

[23] *Amazon web services*, https://aws.amazon.com/, Amazon Web Services, Inc.

[24] *Microsoft azure: Cloud services*, https://azure.microsoft.com/en-us/, Microsoft Corporation.

[25] *Google cloud platform*, https://cloud.google.com/, Google LLC.

[26] R. Prodan and S. Ostermann, "A survey and taxonomy of infrastructure as a service and web hosting cloud providers," in *2009 10th IEEE/ACM International Conference on Grid Computing*, IEEE, 2009, pp. 17–25.

[27] A. Bonci, M. Pirani, and S. Longhi, "A database-centric approach for the modeling, simulation and control of cyber-physical systems in the factory of the future.," *IFAC-PapersOnLine*, vol. 49, no. 12, pp. 249–254, 2016.

[28] J. Lee, B. Bagheri, and H.-A. Kao, "A cyber-physical systems architecture for industry 4.0-based manufacturing systems," *Mfg Letters 3*, 2015.

[29] L. Monostori, "Cyber-physical production systems: Roots from manufacturing science and technology," *Automatisierungstechnik*, vol. 63, no. 10, pp. 766–776, 2015.

[30] C. Snijders, U. Matzat, and U.-D. Reips, "Big data: Big gaps of knowledge in the field of internet science," *International Journal of Internet Science*, vol. 7, no. 1, pp. 1–5, 2012.

[31] M. Little, "Transactions and web services," *Communications of the ACM*, vol. 46, no. 10, pp. 49–54, 2003.

[32] H. Plattner, "A common database approach for oltp and olap using an in-memory column database," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '09, Providence, Rhode Island, USA: Association for Computing Machinery, 2009, pp. 1–2, ISBN: 9781605585512.

[33] S. Chaudhuri and U. Dayal, "An overview of data warehousing and olap technology," *ACM Sigmod record*, vol. 26, no. 1, pp. 65–74, 1997.

[34] J. H. Kim, "A review of cyber-physical system research relevant to the emerging it trends: Industry 4.0, iot, big data, and cloud computing," *Journal of Industrial Integration and Management*, vol. 2, no. 2, 2017.

[35] B. D. Archives, *Database fundamentals – when to use nosql vs sql*, https://starship-knowledge.com/category/big-data, 2020.

[36] M. J. Egenhofer, "Spatial sql: A query and presentation language," *IEEE Transactions on knowledge and data engineering*, vol. 6, no. 1, pp. 86–95, 1994.

[37] M. Stonebraker, "Sql databases v. nosql databases," *Commun. ACM*, vol. 53, no. 4, pp. 10–11, Apr. 2010.

[38] S. Rautmare and D. Bhalerao, "Mysql and nosql database comparison for iot application," *2016 IEEE International Conference on Advances in Computer Applications (ICACA)*, 2016.

[39] H. Fatima and K. Wasnik, "Comparison of sql, nosql and newsql databases for internet of things," *2016 IEEE Bombay Section Symposium (IBSS)*, 2016.

[40] R. Cattell, "Scalable sql and nosql data stores," vol. 39, no. 4, pp. 12–27, May 2011.

[41] R. Housley, W. Ford, W. Polk, and D. Solo, "Internet x. 509 public key infrastructure certificate and crl profile," RFC 2459, January, Tech. Rep., 1999.

[42] I.-L. Yen, S. Zhang, and F. Bastani, "A framework for iot-based monitoring and diagnosis of manufacturing systems," *2017 IEEE Symposium on Service-Oriented System Engineering*, 2017.

[43] *Apache kafka*, Apache Software Foundation, https://kafka.apache.org/.

[44] K. M. M. Thein, "Apache kafka: Next generation distributed messaging system," *International Journal of Scientific Engineering and Technology Research*, vol. 3, no. 47, pp. 9478–9483, 2014.

[45] W. Tärneberg, V. Chandrasekaran, and M. Humphrey, *Experiences creating a framework for smart traffic control using aws iot*, 2016.

[46] *Apache jmeter*, https://jmeter.apache.org/, The Apache Software Foundation.